This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-6-21-87-4.0

Preface

The complete XENIX Reference Manual is actually divided into six parts and distributed as individual reference sections in the various volumes of the XENIX Operating, Text Processing, and Development Systems. The following table lists the name, content, and location of each reference section.

Section	Description	XENIX Volume
C	Commands – used with the XENIX Operating System.	User's Reference
CP	Programming Commands – used with the Development System.	Programmer's Reference
CT	Text Processing Commands – used with the Text Processing System.	Text Processing Guide
DOS	Routines – used with the Development System	Programmer's Reference
F	File Formats – description of various system files not defined in section M.	User's Reference
HW	Hardware specific manual pages – information about XENIX procedures specific to your computer.	Run Time Environment
M	Miscellaneous – information used for access to devices, system maintenance, and communications.	User's Reference
S	System Calls and Library Routines – available for C and assembly language programming.	Programmer's Reference

In the manual pages, a given command, routine, or file is referred to by name and section. For example, the programming command "cc", which is described in the Programming Commands (CP) section, is listed as *cc*(CP).

The alphabetized table of contents given on the following pages is a complete listing of all XENIX commands, system calls, library routines, and file formats. The permuted index, found at the end of the XENIX *User's Reference*, and the the end of the XENIX *Programmer's Reference*, is useful in matching a desired task with the manual page that describes it.

Alphabetized List

Commands, Systems Calls, Library Routines and File Formats

Contents

Programming Commands (CP)

rmdel	Removes a delta from an SCCS file.
sact	Prints current SCCS file editing activity.
sccsdiff	Compares two versions of an SCCS file.
sdb	Invokes symbolic debugger.
size	Prints the size of an object file.
spline	Interpolates smooth curve.
stackuse	Stack requirements for a C program, determines.
strings	Finds the printable strings in an object file.
strip	Removes symbols and relocation bits.
time	Times a command.
tsort	Sorts a file topologically.
unget	Undoes a previous get of an SCCS file.
val	Validates an SCCS file.
xref	Cross-references C programs.
xstr	Extracts strings from C programs.
yacc	Invokes a compiler-compiler.

Name

intro – Introduces XENIX Development System commands.

Description

This section describes use of the individual commands available in the XENIX Development System. Each individual command is labeled with the letters CP to distinguish it from commands available in the XENIX Operating and Text Processing Systems. These letters are used for easy reference from other documentation. For example, the reference *cc*(CP) indicates a reference to a discussion of the **cc** command in this section, where the letter "C" stands for "Command" and the letter "P" stands for "Programming".

Syntax

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

name [*options*] [*cmdarg*]

where:

name The filename or pathname of an executable file

option A single letter representing a command option. By convention, most options are preceded with a dash. Option letters can sometimes be grouped together as in **-abcd** or alternatively they are specified individually as in **-a -b -c -d** . The method of specifying options depends on the syntax of the individual command. In the latter method of specifying options, arguments can be given to the options. For example, the **-f** option for many commands often takes a following filename argument.

cmdarg A pathname or other command argument *not* beginning with a dash. It may also be a dash alone by itself indicating the standard input.

See Also

getopt(C), getopt(S)

Diagnostics

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

Notes

Not all commands adhere to the above syntax.

Name

adb – Invokes a general-purpose debugger.

Syntax

adb [**–w**] [**–p** prompt] [objfil [corefile]]

Description

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

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

Requests to *adb* are read from the standard input and responses are to the standard output. If the **–w** option is present then both *objfil* and *corefile* are created if necessary and opened for reading and writing so that files can be modified using *adb*. The QUIT and INTERRUPT keys cause *adb* to return to the next command. The **–p** option defines the prompt string. It may be any combination of characters. The default is an asterisk (*).

In general requests to *adb* are of the form:

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

If *address* is present then *dot* is set to *address*. Initially *dot* is set to 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *address* is a special expression having the form:

[*segment*:]*offset*

where *segment* gives the address of a specific text or data segment, and *offset* gives an offset from the beginning of that segment. If *segment* is not given, the last segment value given in a command is used.

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

Expressions

. The value of *dot*.

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

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

" The last *address* typed.

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

integer.fraction
A 32-bit floating point number.

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

< *name*
The value of *name*, which is either a variable name or a register name. *adb* maintains a number of variables (see *Variables*) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corefile*. The register names are **ax bx cx dx di si bp fl ip cs ds ss es sp**. The name **fl** refers to the status flags.

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

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

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

Monadic operators

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

-exp Integer negation.

~exp Bitwise complement.

Dyadic operators

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

e1+e2 Integer addition.

e1−e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1^e2 Remainder after division of *e1* by *e2*.

e1#e2 *E1* rounded up to the next multiple of *e2*.

Commands

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

?*f* Locations starting at *address* in *objfil* are printed according to the format *f*.

/*f* Locations starting at *address* in *corefile* are printed according to the format *f*.

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

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

o	2	Prints 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O	4	Prints 4 bytes in octal.
q	2	Prints in signed octal.
Q	4	Prints long signed octal.

d	2	Prints in decimal.
D	4	Prints long decimal.
x	2	Prints 2 bytes in hexadecimal.
X	4	Prints 4 bytes in hexadecimal.
u	2	Prints as an unsigned decimal number.
U	4	Prints long unsigned decimal.
f	4	Prints the 32 bit value as a floating point number.
F	8	Prints double floating point.
b	1	Prints the addressed byte in octal.
c	1	Prints the addressed character.
C	1	Prints the addressed character using the following escape convention. Character values 000 to 040 are printed as an at-sign (@) followed by the corresponding character in the octal range 0100 to 0140. The at-sign character itself is printed as @@.
s	*n*	Prints the addressed characters until a zero character is reached.
S	*n*	Prints a string using the at-sign (@) escape convention. Here *n* is the length of the string including its zero terminator.
Y	4	Prints 4 bytes in date format (see *ctime*(S)).
i	n	Prints as machine instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a	0	Prints the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below. / local or global data symbol ? local or global text symbol = local or global absolute symbol
A	0	Prints the value of *dot* in absolute form.
p	2	Prints the addressed value in symbolic form using the same rules for symbol lookup as **a**.
t	0	When preceded by an integer, tabs to the next appropriate tab stop. For example, 8t moves to the next 8-space tab stop.
r	0	Prints a space.
n	0	Prints a newline.
"..."	0	Prints the enclosed string.
^		Decrements *dot* by the current increment. Nothing is printed.
+		Increments *dot* by 1. Nothing is printed.
–		Decrements *dot* by 1. Nothing is printed.

newline

If the previous command temporarily incremented *dot*, makes the increment permanent. Repeat the previous command with a *count* of 1.

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

[?/]w *value* ...
Writes the 2-byte *value* into the addressed location. If the command is **W**, writes 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]m *segnum fpos size*
Sets new values for the given segment's file position and size. If *size* is not given, then only the file position is changed. The *segnum* must the segment number of a segment already in the memory map (see *Addresses*). If **?** is given, a text segment is affected; if **/** a data segment.

[?/]M *segnum fpos size*
Creates a new segment in the memory map. The segment is given file position *fpos* and physical size *size* . The *segnum* must not already exist in the memory map. If **?** is given, a text segment is created; if **/** a data segment.

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

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

$*modifier*
Miscellaneous commands. The available *modifiers* are:

- <*f* Read commands from the file *f* and return.
- >*f* Send output to the file *f*, which is created if it does not exist.
- **r** Print the general registers and the instruction addressed by ip. *Dot* is set to ip.
- **f** Print the floating registers in single or double length.
- **b** Print all breakpoints and their associated counts and commands.
- **c** C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of **bp**). If **C** is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.
- **e** The names and values of external variables are printed.
- **w** Set the page width for output to *address* (default 80).
- **s** Set the limit for symbol matches to *address* (default 255).
- **o** Sets input and output default format to octal.
- **d** Sets input and output default format to decimal.

x Sets input and output default format to hexadecimal.
q Exit from *adb*.
v Print all non zero variables in octal.
m Print the address map.

:modifier
Manage a subprocess. Available modifiers are:

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

d Delete breakpoint at *address*.

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

R [*arguments*]
Same as the **r** command except that *arguments* are passed through a shell before being passed to to the program. This means shell metacharacters can be used in filenames.

co*s*
The subprocess is continued and signal *s* is passed to it, see *signal*(S). If *address* is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

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

k The current subprocess, if any, is terminated.

Variables

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

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

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

b The base address of the data segment.
d The data segment size.
e The entry point.
m The execution type.
n The number of segments.
s The stack segment size.
t The text segment size.

Addresses

Addresses in *adb* refer to either a location in a file or in actual memory. When there is no current process in memory, *adb* addresses are computed as file locations, and requested text and data are read from the *objfil* and *corefile* files. When there is a process, such as after a **:r** command, addresses are computed as actual memory locations.

All text and data segments in a program have associated memory map entries. Each entry has a unique *segment number*. In addition, each entry has the *file position* of that segment's first byte, and the *physical size* of the segment in the file. When a process is running, a segment's entry has a *virtual size* which defines the size of the segment in memory at the current time. This size can change during execution.

When a address is given and no process is running, the file location corresponding to the address is calculated as:

effective-file-address = file-position + offset

If a process is running, the memory location is simply the offset in the given segment. These addresses are valid if and only if

0 <= offset <= size

where *size* is physical size for file locations and virtual size for memory locations. Otherwise, the requested *address* is not legal.

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

So that *adb* may be used on large files, all appropriate values are kept as signed 32 bit integers.

Files

a.out
core

See Also

ptrace(S), a.out(F), core(F)

Diagnostics

The message "adb" appears when there is no current command or format.

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

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

Notes

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

System calls cannot be single stepped.

Local variables whose names are the same as an external variable may foul up the accessing of the external.

Name

admin – Creates and administers SCCS files.

Syntax

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

Description

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

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If the dash **−** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, nonSCCS files and unreadable files are silently ignored.

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

−n This option indicates that a new SCCS file is to be created.

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

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

−f*flag* This option specifies a *flag*, and possibly a value for the *flag*, to be placed in the SCCS file. Several f options may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

b Allows use of the **−b** option on a *get*(CP) command to create branch deltas.

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

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

d*SID* The default delta number (SID) to be used by a *get*(CP) command.

i Causes the "No id keywords (ge6)" message issued by *get*(CP) or *delta*(CP) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(CP)) are found in the text retrieved or stored in the SCCS file.

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

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

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

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

n Causes *delta*(CP) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file preventing branch deltas from being created from them in the future.

q*text* User-definable text substituted for all occurrences of the keyword in SCCS file text retrieved by *get*(CP).

m*mod* *mod*ule name of the SCCS file substituted for all occurrences of the admin.CP keyword in SCCS file text retrieved by *get*(CP). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *type* of module in the SCCS file substituted for all occurrences of keyword in SCCS file text retrieved by *get*(CP).

v[*pgm*] Causes *delta*(CP) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program (see *delta*(CP)). (If this flag is set when creating an SCCS file, the **m** option must also be used even if its value is null).

−d[*flag*] Causes removal (deletion) of the specified *flag* from an SCCS file. The **−d** option may be specified only when processing existing SCCS files. Several **−d** options may be supplied on a single *admin* command. See the **−f** option for allowable *flag* names.

l*list* A *list* of releases to be "unlocked". See the **−f** option for a description of the **l** flag and the syntax of a *list*.

−a*login* A *login* name, or numerical XENIX group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** options may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

−e*login* A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** options may be used on a single *admin* command line.

−y[*comment*] The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(CP). Omission of the **−y** option results in a default comment line being inserted in the form:

YY/*MM*/*DD HH*:*MM*:*SS* by *login*

The **−y** option is valid only if the **−i** and/or **−n** options are specified (i.e., a new SCCS file is being created).

−m[*mrlist*] The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(CP). The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.

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

This option inhibits writing on the file, nullifying the effect of any other options supplied, and is therefore only meaningful when processing existing files.

−z The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see **−h**, above).

Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

Files

The last component of all SCCS filenames must be of the form **s.***file-name*. New SCCS files are created read-only (444 modified by umask) (see *chmod*(C)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called **x.***filename*, (see *get*(CP)), created with read-only permission if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be read-only. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of a text editor. *Care must be taken!* The edited file should *always* be processed by an **admin −h** to check for corruption followed by an **admin −z** to generate a proper checksum. Another **admin −h** is recommended to ensure the SCCS file is valid.

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

See Also

delta(CP), ed(C), get(CP), help(CP), prs(CP), what(C), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Name

ar – Maintains archives and libraries.

Syntax

ar key [posname] afile name ...

Description

ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor though it can be used for any similar purpose.

key is one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcln**. *afile* is the archive file. The *names* are constituent files in the archive file. The *posname* is the name of a constituent file, and is required when certain keys are used. The meanings of the *key* characters are:

d Deletes the named files from the archive file.

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

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

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

p Prints the named files in the archive.

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

x Extracts the named files. If no names are given, all files in the archive are extracted. Unless the optional character n is used with x, an extracted file's modification date will be set to the date stored in that file's archive header. In neither case does **x** alter the archive file.

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

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

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

n New. When used with the *key* character **x** it sets the extracted file's modification date to the current date.

When *ar* creates an archive, it always creates the header in the format of the local system (see *ar*(F)).

Files

/tmp/v* Temporary files

See Also

ld(CP), lorder(CP), ar(F)

Notes

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

Failure to process a library with *ranlib*, or failure to reprocess a library with *ranlib*, will cause *ld* to fail. Because generation of a library by *ar* and randomization by *ranlib* are separate, phase errors are possible. The loader *ld* warns when the modification date of a library is more recent than the creation of its dictionary; but this means you get the warning even if you only copy the library.

Name

asx – XENIX 8086/186/286/386 assembler.

Syntax

asx [*options*] *source-file*

Description

asx assembles 8086/186/286/386 assembly language source files and produces linkable object modules. Note that *masm*(CP) is the supported XENIX assembler and should be used instead of *asx* for new development.

asx accepts one *source-file*. The source file name must have the ".s" extension. The resulting file containing the object module is given the same base name as the source, with the ".o" extension replacing the ".s" extension.

There are the following options:

-a Assembled segments are output in alphabetic order, instead of in order of occurrence in the source file.

-d Creates program listings for both passes of the assembler. This listing can be used to resolve phase errors between assembler passes. The **-d** option is ignored if the **-l** option is not in effect.

-l Produces a listing file. The listing file has the same base name as the source file, but has the ".lst" extension.

-Mu Disables case sensitivity for all names and symbols. This option makes upper and lowercase letters in names and symbols indistinguishable to the assembler. This option also causes the symbols defined by the EXTRN and PUBLIC directives to be output in uppercase regardless of their original spelling.

-Mx Disables case sensitivity for all names and symbols except those names defined by the EXTRN and PUBLIC directives. This option is similar to the **-Mu** option except that public and external names copied to the object file retain their original spelling.

-n Suppresses the generation of the symbol table in the program listing. This option is ignored if the **-l** option is not in effect.

-o *filename*
Directs the generated object module to the file named *filename*. No default extension is assumed.

-O Causes values in the program listing to be displayed in octal. The default radix is hexadecimal.

-r Causes generation of actual 8087/287 instructions instead of software interrupts for the floating point emulation package. Object modules created using this option can only be executed on machines with an 8087 or 287.

-X Directs the assembler to list any conditional block whose IF condition resolves to false. This option can be overridden in the source file by using the .TFCOND directive. This option is ignored if the -l option is not in effect.

By default, *asx* recognizes 8086 instruction mnemonics only. To assemble 186, 286, 386, 8087, or 287 instructions, the corresponding .186, .286c, .286p, .386, .8087, or .287 directive must be given in the source file.

Files

/bin/asx

See Also

ld(CP)

Note

Unless the **-r** is given, **asx** assumes all 8087/287 instructions are to be carried out using floating point emulation. The **-r** option should only be used on machines with an 8087 or 287 coprocessor.

asx (CP) is also known as the Ritchie assembler. It was used before the introduction of the cmerge C compiler and is not compatible with **cc** (CP). Use **ld**(CP) to link object modules created with **asx**.

Name

cb – Beautifies C programs.

Syntax

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

Description

cb places a copy of the C program in *file* (standard input, if *file* is not given) on the standard output with spacing and indentation that displays the structure of the program. Under default options, *cb* preserves all user newlines. The **-**s option formats the code to match the style of Kernighan and Ritchie in *The C Programming Language.* The **-j** option causes split lines to be put back together. The **-l** option causes *cb* to split lines that are longer than *leng*.

See Also

cc(CP)

B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (Englewood Cliffs: Prentice-Hall, 1978)

Notes

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

Name

cc – Invokes the C compiler.

Syntax

cc [option ...] filename ...

Description

cc is the XENIX C compiler command. It creates executable programs by compiling and linking the files named by the *filename* arguments. *cc* copies the resulting program to the file **a.out**.

The *filename* can name any C or assembly language source file or any object or library file. C source files must have a **.c** filename extension. Assembly language source files must have **.s**, object files **.o**, and library files **.a** extensions. *cc* invokes the C compiler for each C source file and copies the result to an object file whose basename is the same as the source file but whose extension is **.o**. *cc* invokes the XENIX assembler, *masm* , for each assembly source file and copies the result to an object file with extension **.o**. *cc* ignores object and library files until all source files have been compiled or assembled. It then invokes the XENIX link editor, *ld* , and combines all the object files it has created together with object files and libraries given in the command line to form a single program.

Files are processed in the order they are encountered in the command line, so the order of files is important. Library files are examined only if functions referenced in previous files have not yet been defined. Library files must be in *ranlib*(CP) format, that is, the first member must be named __.SYMDEF, which is a dictionary for the library. Only those functions that define unresolved references are concatenated. A number of "standard" libraries are searched automatically. These libraries support the standard C library functions and program startup routines. Which libraries are used depends on the program's memory model (see "Memory Models" below). The entry point of the resulting program is set to the beginning of the standard startup code which then calls the "main( )" function of the program.

There are the following options:

-c
Creates a linkable object file for each source file but does not link these files. No executable program is created.

-C
Preserves comments when preprocessing a file with -**E** , -**P** , or -**EP**. That is, comments are not removed from the

preprocessed source. This option may only be used in conjunction with **-E** , **-P** , or **-EP**.

-compat
Makes an executable file that is binary compatible across the following systems (as distributed by certain vendors):

XENIX-286 System V
XENIX-386 System V
XENIX-286 3.0
XENIX-8086 System V

-CSON, -CSOFF
When optimization (**-O**) is also specified, these options enable or disable "common sub-expression" optimization. The default is disabled for the small model passes and enabled for the large (with **-LARGE**).

-d Displays the various passes and their arguments before they are executed.

-D*name*[=*string*]
Defines *name* to the preprocessor as if defined by #define in each source file. The form "**-D***name*" sets *name* to 1. The form "**-D***name*=*string*" sets *name* to the given *string*.

-dos
Directs **cc** to create an executable program for MS-DOS systems.

-E Preprocesses each source file as described for **-P**, but copies the result to the standard output. The option also places a #line directive with the current input line number and source file name at the beginning of output for each file.

-EP
Preprocesses each source file as described for **-E**, but does not place a #line directive at the beginning of the file.

-F *num*
Sets the size of the program stack to *num* bytes. The value of *num* must be given in hexadecimal. The default stack for the 8086 is variable, starting at the top of a full 64 Kbyte data segment that grows down until it reaches data. The default stack for the 80286 is 1000 bytes (hexadecimal). This option does not apply to the 80386, which has a variable stack.

-Fa, **-Fa***name*
Create an assembly source listing in source.s or the named file. Continues with the link if requested.

-Fc, **-Fc***name*
Create a merged assembler and C listing in source.L or in the named file.

-Fe*name*
Names the executable program file *name*.

-Fl, **-Fl***name*
Create a listing file in source.L (or the named file) with assembly source and object code. Continues with the link if requested.

-Fm, **-Fm***name*
Instruct the linker to create a map listing in a file called a.map (or the named file). This file contains the names of all segments in order of their appearance in the load module.

-Fo*name*
The object filename will be *name* instead of source.o.

-FPa, **-FPc**, **-FPc87**, **-FPi**, **-FPi87**
When used in conjunction with **-dos** these options control the type of floating point code generated and which library support to use. The default is **-FPi**. For more information see Appendix A, "XENIX to DOS: A Cross Development System" , of the XENIX *C Library Guide* .

-Fs, **-Fs***name*
Creates a C source listing in source.S or the named file.

-g
Includes information for the symbolic debugger. (This is equivalent to the **-Z***i* option.)

-i
Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and may be shared by all users executing the file. This option is implied when creating middle or large model programs. (Not implemented on all machines.)

-I*pathname*
Adds *pathname* to the list of directories to be searched when an #include file is not found in the directory containing the current source file or whenever angle brackets (< >) enclose the filename. If the file cannot be found in directories in this list, directories in a standard list are searched.

-K
Removes stack probes from a program. Stack probes are used to detect stack overflow on entry to program routines. Code

generated for the 80386 processor does not require stack probes, therefore this option has no effect if -M3 is specified.

-l*name*
Searches library *name* for unresolved function references.

-L Creates an assembler listing file containing assembled code and assembly source instructions. The listing is made in a file whose basename is the same as the source but whose extension is .L. This option suppresses the -S option.

-LARGE
Invokes the large model passes of the compiler (executable on 286 and 386 processors only). Using large model passes is advised when "Out of heap space" errors are encountered.

-M *string*
Sets the program configuration. This configuration defines the program's memory model, word order, and data threshold. It also enables C language enhancements such as advanced instruction set and keywords. The *string* may be any combination of the following ("s", "m", "l", and "h" are mutually exclusive):

s	Creates a small model program (default).
m	Creates a middle model program.
l	Creates a large model program.
h	Creates a huge model program.
e	Enables the far, near, huge, pascal, and fortran keywords. Also enables certain non-ANSI entensions necessary to ensure compatibility with existing versions of the C compiler (applies only to versions of the C compiler that support ANSI C).
0	Enables 8086 code generation for compiled C source files. Default is 8086 code generation.
1	Enables 186 code generation for compiled C source files.
2	Enables 286 code generation for compiled C source files.
3	Enables 386 code generation for compiled C source files (80386 processors only).
b	Reverses the word order for long types. High order word is first. Default is low order word first.
t*num*	Causes all data items greater than *num* bytes to be allocated to a new data segment. *Num*, the data threshold, defaults to 32,767. This option can only be used in large model 8086/80286 programs (Ml0 or Ml2).
d	Instructs the compiler to not assume SS=DS. Warning: This option has no practical use on XENIX. It will *not* cause the stack to be put in a separate segment. It may be used for DOS cross development.

-n Sets pure text model. This option is equivalent to the **-i** option. Gives a warning that it is setting **-i** when used.

-ND *name*
Sets the data segment name for each compiled or assembled source file to *name*. If -**ND** is not given, the name "_DATA" is used.

In large model programs (-**Ml**) the -**ND** option can only be used on "leaf modules" – those that make no calls to routines in another segment.

-nl *num*
Sets the maximum length of external symbols to *num*. Names longer than *num* are truncated before being copied to the external symbol table.

-NM *name*
Sets the module name for each compiled or assembled source file to *name*. If not given, the filename of each source file is used.

-NT *name*
Sets the text segment name for each compiled or assembled source file to *name*. If not given, the name "*module*_TEXT" is used for middle model and "_TEXT" for small model programs. This option should not be used on 386 code.

-o *filename*
Defines *filename* to be the name of the final executable program. This option overrides the default name **a.out**. *Filename* can not end in **.o** or **.c**.

-O *string*
Invokes the object code optimizer. The *string* consists of one or more of the following characters:

d	Default. Disables optimization
a	Relaxes alias checking
s	Optimizes code for space
t	Default. Optimizes code for speed. Equivalent to -O
x	Performs maximum optimization. Equivalent to **-Oactl**
c	Eliminates common expressions
l	Performs various loop optimizations.

-p
Adds code for program profiling. Profiling code counts the number of calls to each routine in the program and copies this information to the **mon.out** file. This file can be examined using the *prof*(CP) command.

-P
Preprocesses each source file and copies the result to a file whose basename is the same as the source but whose extension is **.i**.

-pack
Packs structures. Each structure member is stored at the first available byte, without regard to *int* boundaries. Although this will save space, execution will be slower because of the extra time required to access 16 bit members that begin on odd boundaries.

-r Invokes the incremental linker, **/lib/ldr** , for the link step.

-s Instructs the linker to strip the symbol table information from the executable output file.

-S
Creates an assembly source listing in a file whose basename is the same as the source but whose extension is **.s**. It should be noted that this file is not suitable for assembly. This option provides code for reading only.

-SEG *num*
Sets the maximum number of segments that the linker can handle to *num*, which can range from 1 to 1024. If 1024 is too small, use the **-NT** option to reduce the number of different segment names.

-u Eliminates all manifest defines. Also see **-U**.

-U *definition*
Removes or undefines the given manifest define. The manifest defines are as follows:

M_I86
M_XENIX
M_SYS3 or M_SYSIII
M_SYS5 or M_SYSV
M_BITFIELDS
M_WORDSWAP
M_SDATA or M_LDATA
M_STEXT or M_LTEXT
M_I8086 or M_I186 or M_I286 or M_I386
M_I86SM or M_I86MM or M_I86LM

-V *string*
Copies *string* to the object file created from the given source file. This option can be used for version control.

-w Prevents compiler warning messages from being issued. Same as "-W 0".

-W *num*
Sets the output level for compiler warning messages. If *num* is 0, no warning messages are issued. If 1, only warnings about program structure and overt type mismatches are issued. If 2, warnings about strong typing mismatches are issued. If 3, warnings for all automatic conversions are issued. This option does not affect compiler error message output.

-X
Removes the standard directories from the list of directories to be searched for #include files.

-z Displays the various passes and their arguments but does not execute them.

-Zp1, -Zp2, -Zp4
Aligns data structures on one, two or four-byte boundaries (80386 only).

-Zi
Includes information used by the symbolic debugger (sdb) in the output file. (This is equivalent to the **-g** option.)

Many options (or equivalent forms of these options) are passed to the link editor as the last phase of compilation. The **-M** option with the "s", "m", and "l" configuration options are passed to specify memory requirements. The **-i**, **-F**, and **-p** are passed to specify other characteristics of the final program.

The **-D** and **-I** options may be used several times on the command line. The **-D** option must not define the same name twice. These options affect subsequent source files only.

Memory Models

cc can create programs for four different memory models: small, middle, large, and huge. In addition, small model programs can be pure or impure. On the 8086 and 80286 processors, these various segmentation models allow programs with code or data larger than 64K bytes. Since the 80386 can address segments larger than 64K bytes, the middle, large and huge models are not supported on the 80386.

Impure-Text Small Model

These programs occupy one 64K byte physical segment in which both text and data are combined. *cc* creates impure small model programs by default. They can also be created using the -Ms option.

Pure-Text Small Model

These programs occupy two 64K byte physical segments. Text and data are in separate segments. The text is read-only and may be shared by several processes at once. The maximum program size is 128 Kbytes. Pure small model programs are created using the -i and -Ms options.

Middle Model

These programs occupy several physical segments, but only one segment contains data. Text is divided among as many segments as required. Special calls and returns are used to access functions in other segments. Text can be any size. Data must not exceed 64K bytes. Middle models programs are created using the -Mm option. These programs are always pure.

Large Model

These programs occupy several physical segments with both text and data in as many segments as required. Special calls and returns are used to access functions in other segments. Special addresses are used to access data in other segments. Text and data may be any size, but no data item may be larger than 64K bytes. Large model programs are created using the -Ml option. These programs are always pure.

Huge Model

These programs occupy several physical segments with both text and data in as many segments as required. It is possible to allow a data construct that spans 64K byte segments. This implementation imposes limits on the way the data construct is put together and where it is located in memory. Huge model programs are created using the -Mh option. These programs are always pure.

Small, middle, large and huge model object files can only be linked with object and library files of the same model. It is not possible to combine small, medium, large, and huge model object files in one executable program. *cc* automatically selects the correct small, middle, large, or huge versions of the standard libraries based on the configuration option. It is up to users to make sure that all of their own object files and private libraries are properly compiled in the appropriate model.

The special calls and returns used in middle, large, and huge model programs may affect execution time. In particular, the execution time of a program which makes heavy use of functions and function pointers may differ noticably from small model programs.

In middle, large, and huge model programs, function pointers are 32 bits long. In large and huge model programs, data pointers are 32 bits long. Programs making use of such pointers must be written carefully to avoid incorrect declaration and use of these variables.

The **-NM**, **-NT**, and **-ND** options may be used with middle, large, and huge model programs to direct the text and data of specific object files to named physical segments. All text having the same text segment name is placed in a single physical segment. Similarly, all data having the same data segment name is placed in a single physical segment.

cc reads **/etc/default/cc** to obtain information about default options and libraries. The default file may contain lines beginning with the following patterns:

FLAGS=

and

LIBS=

Any parameters following the FLAGS= pattern are treated by *cc* as if they had been specified at the start of the *cc* command line. Parameters following the LIBS= pattern are treated as if they had been specified at the end of the command line. This option is intended for, but not restricted to, the specification of additional libraries. *cc* always searches for a file in **/etc/default** that matches the last component of the pathname by which *cc* was invoked. Thus by linking *cc* to several different names and invoking it by those names, different defaults can be selected.

An example **/etc/default/cc** file follows:

FLAGS= –LARGE –M2e

LIBS= –lx

This invokes the large model versions of the compiler passes to generate 286 code with **far** and **near** keywords enabled, and includes **libx.a** on all links.

Files

/bin/cc	Driver
/lib/p0, p1, p2, p3	Small model passes
/lib/p1L, p2L, p3L	Large model passes
/lib/*.a	Standard libraries
/etc/default/cc	Default options and libraries

See Also

ar(CP), ld(CP), lint(CP), machine(M), masm(CP), ranlib(CP)
XENIX C User's Guide, *C Library Guide*, and *C Language Reference*

Notes

Error messages are produced by the program that detects the error. These messages are usually produced by the C compiler, but may occasionally be produced by the assembler or the link loader.

All object module libraries must have a current *ranlib* directory. The user must make sure that the most recent library versions have been processed with *ranlib*(CP) before linking. If this is not done, *ld* cannot create executable programs using these libraries.

Name

cdc – Changes the delta commentary of an SCCS delta.

Syntax

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

Description

cdc changes the delta commentary for the *SID* specified by the **−r** option, of each named SCCS file.

delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the *delta*(CP) command (**−m** and **−y** options).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of **−** is given, the standard input is read (see Warning); each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of options and file names.

All the described options apply independently to each named file:

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

−m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(CP)) then a list of MR numbers to be added and/or deleted in the delta commentary of the *SID* specified by the **−r** option *may* be supplied. A null MR list has no effect.

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

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

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

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

−y[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **−r** option. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

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

In general, if you made the delta, you can change its delta commentary; or if you own the file and directory you can modify the delta commentary.

Examples

The following:

```
cdc −r1.6 −m"bl78-12345 !bl77-54321 bl79-00001" −ytrouble
s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of s.file.

The following interactive sequence does the same thing.

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

Warning

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

Files

x-file See *delta*(CP)

z-file See *delta*(CP)

See Also

admin(CP), delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Name

cflow – Generates C flow graph.

Syntax

cflow [**-r**] [**-ix**] [**-i_**] [**-d**num] file ...

Description

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

Each line of output begins with a line number, followed by a suitable number of tabs indicating the level, the name of the global procedure, a colon, and the definition. A global procedure is normally a function not defined as an external and not beginning with an underscore character (see the **-i** option on the next page). For information extracted from C source files, the definition includes an abstract type declaration (for example, **char ***), and, enclosed by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (for example, *text*). Leading underscores in C-style external names are deleted.

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

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

```
int	i;

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

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

the command:

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

produces the following C flow graph:

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

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

The following options are interpreted by *cflow*:

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

-ix Includes external and static data symbols. The default is to include only functions in the flow graph.

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

-dnum Indicates the depth (*num* decimal integer) at which the flow graph is cut off. By default this is a very large number. You can not set the cutoff depth to a nonpositive integer.

See Also

cc(CP), lex(CP), lint(CP), masm(CP), nm(CP), pr(C), yacc(CP)

Diagnostics

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

Notes

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

Name

comb – Combines SCCS deltas.

Syntax

comb [**−o**] [**−s**] [**−psid**] [**−clist**] files

Description

comb provides the means to combine one or more deltas in an SCCS file and make a single new delta. The new delta replaces the previous deltas, making the SCCS file smaller than the original.

comb does not perform the combination itself. Instead, it generates a shell procedure that you must save and execute to reconstruct the given SCCS files. *comb* copies the generated shell procedure to the standard output. To save the procedure, you must redirect the output to a file. The saved file can then be executed like any other shell procedure (see *sh*(C)).

When invoking *comb*, arguments may be specified in any order. All options apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of **−** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; nonSCCS files and unreadable files are silently ignored.

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

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

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

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

 COMB (CP)

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

100 * (original - combined) / original

Before any SCCS files are actually combined, you should use this option to determine exactly how much space is saved by the combining process.

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

Files

comb????? Temporary files

See Also

admiu(CP), delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Notes

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

Name

cpp – The C language preprocessor.

Syntax

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

Description

cpp is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(CP) command. Thus the output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(CP) command. See *m4*(CP) for a general macro processor.

cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

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

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

–U*name*
: Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.

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

–I*dir*
: Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" are searched for first in the directory of the *ifile* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files

whose names are enclosed in <>, the directory of the *ifile* argument is not searched.

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

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

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

#define *name*(*arg*, ..., *arg*) *token-string*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma separated tokens, and a) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma separated list.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on.

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

#line *integer-constant* "*filename*"
Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If "*filename*" is not given, the current file name is unchanged.

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

#ifdef *name*
The following lines appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
The following lines will not appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#if defined *identifier*

May be used in place of the **#if** directive. If the *identifier* is defined, the directive has a value of 1, otherwise 0. This is frequently used for conditional environment-specific text.

#elif *constant-expression*

Allows for the conditional compilation of portions of the text. The *constant-expression* is evaluated and if it is not zero, the text immediately following (until the next **elif, else, endif**) is passed to the compiler.

#if *constant-expression*

The following lines appear in the output if *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the ?: operator, the unary −, !, and ˜ operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined (** *name* **)** or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

#else

Reverses the notion of the test directive which matches this directive. So if lines previous to this directive are ignored, the following lines appear in the output. And vice versa.

The test directives and the possible **#else** directives can be nested.

Files

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

See Also

cc(CP), m4(CP).

Diagnostics

The error messages produced by *cpp* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

Notes

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

Name

cref – Makes a cross-reference listing.

Syntax

cref [**−acilnostux123**] files

Description

cref makes a cross-reference listing of assembler or C programs. The program searches the given *files* for symbols in the appropriate C or assembly language syntax.

The output report is in four columns:

1. Symbol
2. Filename
3. Current symbol or line number
4. Text as it appears in the file

cref uses either an *ignore* file or an *only* file. If the **−i** option is given, the next argument is taken to be an *ignore* file; if the **−o** option is given, the next argument is taken to be an *only* file. *ignore* and *only* files are lists of symbols separated by newlines. All symbols in an *ignore* file are ignored in columns 1 and 3 of the output. If an *only* file is given, only symbols in that file will appear in column 1. Only one of these options may be given; the default setting is **−i** using the default ignore file (see *FILES* below). Assembler predefined symbols or C keywords are ignored.

The **−s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The **−l** option causes the line number within the file to be put in column 3.

The **−t** option causes the next available argument to be used as the name of the intermediate file (instead of the temporary file **/tmp/crt??**). This file is created and is *not* removed at the end of the process.

The *cref* options are:

a Uses assembler format (default)

c Uses C format

i Uses an *ignore* file (see above)

l Puts line number in column 3 (instead of current symbol)

n Omits column 4 (no context)

o Uses an *only* file (see above)

s Current symbol in column 3 (default)

t User-supplied temporary file

u Prints only symbols that occur exactly once

x Prints only C external symbols

1 Sorts output on column 1 (default)

2 Sorts output on column 2

3 Sorts output on column 3

Files

/usr/lib/cref/* Assembler specific files

See Also

as(CP), cc(CP), sort(C), xref(CP)

Notes

cref inserts an ASCII DEL character into the intermediate file after the eighth character of each name that is eight or more characters long in the source file.

Name

ctags – Creates a tags file.

Syntax

ctags [**–a**] [**–u**] [**–v**] [**–w**] [**–x**] name ...

Description

ctags makes a tags file for *vi*(C) from the specified C sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the filc in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *vi* can quickly find these function definitions.

If the **–x** flag is given, *ctags* produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. With the **–x** option no tags file is created. This is a simple index which can be printed out as an off-line readable function index.

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions.

Other options are:

–w Suppresses warning diagnostics.

–u Causes the specified files to be *updated* in tags; that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

Files

tags	Output tags file

See Also

ex(C), vi(C)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Name

cxref – Generates C program cross-reference.

Syntax

cxref [options] file ...

Description

cxref analyzes a collection of C files and attempts to build a cross-reference table. *cxref* uses a special version of *cpp* to include information defined by **#define** in its symbol table. It produces a listing on the standard output of all symbols (auto, static, and global) for each separate file, or with the **-c** option for the combined files. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **-D**, **-I** and **-U** options (which are identical to their interpretation by *cc*(CP)), the following *options* are interpreted by *cxref*:

-c	Prints a combined cross-reference of all input files.
-w<*num*>	Formats output no wider than <*num*> (decimal) columns. The default is 80 if <*num*> is not specified or is less than 51.
-o *file*	Directs output to named *file*.
-s	Operates silently; does not print input filenames.
-t	Formats listing for 80-column width.

Files

/usr/lib/xcpp   special version of C-preprocessor.

See Also

cc(CP)

Diagnostics

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

Notes

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

Name

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

Syntax

delta [**−**rSID] [**−s**] [**−n**] [**−g**list] [**−m**[mrlist]] [**−y**[comment]] [**−p**] files

Description

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

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

delta may issue prompts on the standard output depending upon certain options specified and flags (see *admin*(CP)) that may be present in the SCCS file (see **−m** and **−y** options below).

Options apply independently to each named file.

−r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more versions of the same SCCS file have been retrieved for editing (**get −e**) by the same person (login name). The SID value specified with the **−r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(CP)). A diagnostic results if the specified SID is ambiguous, or if it is necessary and omitted on the command line.

−s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

−n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

-g*list* Specifies a *list* (see *get*(CP) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

-m[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(CP)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

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

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

Note that if the **v** flag has a value (see *admin*(CP)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, *delta* terminates (it is assumed that the MR numbers were not all valid).

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

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

-p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied. Differences are displayed in a *diff*(C) format.

Files

All files of the form ?-file are explained in Chapter 3, "SCCS: A Source Code Control System" in the *XENIX Programmer's Guide*. The naming convention for these files is also described there.

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

Warning

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

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (**−**) is specified on the *delta* command line, the **−m** (if necessary) and **−y** options *must* also be present. Omission of these options causes an error to occur.

See Also

admin(CP), bdiff(C), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Name

dosld – XENIX to MS–DOS cross linker

Syntax

dosld *options* file ...

Description

dosld links the object files(s) given by *file* to create a program for execution under MS–DOS. Although similar to *ld*(CP), *dosld* has many options that differ significantly from *ld*. The options are described below:

–D
DS Allocate. This instructs *dosld* to perform DS allocation. It is generally used in conjunction with the **–H** option.

–H
Load high. This option instructs *dosld* to set a field in the header of the executable file to tell MS–DOS to load the program at the highest available position in memory. It is most often used with programs in which data precedes code in the memory image.

–L
Include line numbers. This option instructs *dosld* to include line numbers in the listing file (if any). Note that *dosld* cannot put line numbers in the listing file if the source translator hasn't put them in the object file.

–M
Include public symbols. This option instructs *dosld* to include public symbols in the list file. The symbols are sorted twice, lexicographically and by address.

–C
Ignore case. This option instructs *dosld* to treat upper and lower case characters in symbol names as identical.

–F *num*
Set stack size. This option should be followed by a hexadecimal number. *dosld* will use this number for the size in bytes of the stack segment in the output file.

–S *num*
Set segment limit. This option should be followed by a decimal number between 1 and 1024. The number sets the limit on the number of different segments that may be linked together. The

default is 128. Note that the higher the value given, the slower the link will be.

−m *filename*
Create map file. This option should be followed by a filename. *dosld* will create a file with the given name in which it will put information about the segments and groups in the executable. Additionally, public symbols and line numbers will be listed in this file if the **−M** and **−L** options are given.

−nl *num*
Set name length. This option should be followed by a decimal number. The option instructs *dosld* to truncate all public and external symbols longer than *num* characters.

−o *filename*
Name output file. This option should be followed by a filename which *dosld* will use as the name of the executable file it creates. The default name is **a.out**.

−u *name*
Name undefined symbol. This option should be followed by a symbol name. *dosld* will enter the given name into its symbol table as an undefined symbol. The **−u** option may appear more than once on the command line.

−G
Ignore group associations. This option instructs *dosld* to ignore any group definitions it may find in the input files. This option is provided for compatibility with old versions of MS−LINK; generally, it should never be used.

As with *ld*, the files passes to *dosld* may be either XENIX-style libraries (objects collected using *ar*(CP) and indexed using *ranlib*(CP)) or ordinary 8086 object files. Unless the **−u** option appears, at least one of the files passed to *dosld* must be an ordinary object file. Libraries are searched only after all the ordinary object files have been processed.

Files

/usr/bin/dosld

See Also

ar(CP), as(CP), cc(CP), ld(CP), ranlib(CP)

Name

get - Gets a version of an SCCS file.

Syntax

get [-**r**SID] [-**c**cutoff] [-**i**list] [-**x**list] [-**a**seq-no.] [-**k**] [-**e**] [-**l**[p]] [-**p**] [-**m**] [-**n**] [-**s**] [-**b**] [-**g**] [-**t**] file ...

Description

get generates an ASCII text file from each named SCCS file according to the specifications given by its options, which begin with **-**. The arguments may be specified in any order, but all options apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of **-** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, nonSCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS filename by simply removing the leading s.; (see also *Files*).

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any option apply independently to each named file.

-r*SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved.

-c*cutoff* *cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **-c7502** is equivalent to **-c750228235959**. Any number of nonnumeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows you to specify a *cutoff* date in the form: "-c77/2/2 9:22:25".

-e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(CP). The **-e** option used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(CP)). Concurrent use of **get -e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **-e** option is accidentally ruined in the editing process, it may be regenerated by reexecuting the *get* command with the **-k** option in place of the **-e** option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(CP)) are enforced when the **-e** option is used.

-b Used with the **-e** option to indicate that the new delta should have an SID in a new branch. This option is ignored if the **b** flag is not present in the file (see *admin*(CP)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)

Note: A branch *delta* may always be created from a nonleaf *delta*.

-i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID - SID

SID, the SCCS Identification of a delta, may be in any form described in the SCCS chapter in the XENIX *Programmer's Guide*.

-x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **-i** option for the *list* format.

-k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **-k** option is implied by the **-e** option.

-l[p] Causes a delta summary to be written into an *l-file*. If **-lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *Files* for the format of the *l-file*.

-p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the -s option is used, in which case it disappears.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **-m** and **-n** options are used, the format is: %M% value, followed by a horizontal tab, followed by the -m option generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

-t Used to access the most recently created (top) delta in a given release (e.g., **-r1**), or release and level (e.g., **-r1.2**).

-a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(F)). This option is used by the *comb*(CP) command; it is not particularly useful and should be avoided. If both the **-r** and **-a** options are specified, the **-a** option is used. Care should be taken when using the **-a** option in conjunction with the **-e** option, as the SID of the delta to be created may not be what yon expect. The **-r** option can be used with the **-a** and **-e** options to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **-e** option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each filename is printed (preceded by a newline) before it is processed. If the **-i** option is used included deltas are listed following the notation "Included"; if the **-x** option is used, excluded deltas are listed following the notation "Excluded".

Identification Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the **m** flag in the file (see *admin* (CP)), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file (see *admin*(CP)).
%F%	SCCS filename.
%P%	Fully qualified SCCS filename.
%Q%	The value of the **q** flag in the file (see *admin* (CP)).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by *what*(C).
%W%	A shorthand notation for constructing *what*(C) strings for XENIX program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(C) strings for nonXENIX program files. %A% = %Z%%Y% %M% %I%%Z%

Files

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary filename is formed from the SCCS filename: the last component of all SCCS filenames must be of the form s.*module-name*, the auxiliary files are named by replacing the leading s with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary filenames would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **-p** option is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the **-k** option is used or implied, the *g-file*'s mode is 644; otherwise the mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **-l** option is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included
   "X": Excluded
   "C": Cut off (by a **-c** option)
d. Blank
e. SCCS identification (SID)
f. Tab character
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation
h. Blank
i. Login name of person who created *delta*

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **-e** option along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **-e** option for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(CP)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **-i** option if it was present, followed by a blank and the **-x** option if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

See Also

admin(CP), delta(CP), help(CP), prs(CP), what(C), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Notes

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the **-e** option is used.

Name

gets – Gets a string from the standard input.

Syntax

gets [string]

Description

gets can be used with *csh*(C) to read a string from the standard input. If *string* is given it is used as a default value if an error occurs. The resulting string (either *string* or as read from the standard input) is written to the standard output. If no *string* is given and an error occurs, *gets* exits with exit status 1.

See Also

line(C), csh(C)

Name

hdr – Displays selected parts of executable binary files.

Syntax

hdr [–dhprsSt] file ...

Description

hdr displays executable binary file headers, symbol tables, and text or data relocation records in human-readable formats. It also prints out seek positions for the various segments in the executable binary file.

a.out, **x.out**, and **x.out** segmented formats and archives are understood.

The symbol table format consists of six fields. In **a.out** formats the third field is missing. The first field is the symbol's index or position in the symbol table, printed in decimal. The index of the first entry is zero. The second field is the type, printed in hexadecimal. The third field is the **s_seg** field, printed in hexadecimal. The fourth field is the symbol's value in hexadecimal. The fifth field is a single character which represents the symbol's type as in *nm*(CP), except **C** common is not recognized as a special case of undefined. The last field is the symbol name.

If long form relocation is present, the format consists of six fields. The first is the descriptor, printed in hexadecimal. The second is the symbol ID, or index, in decimal. This field is used for external relocations as an index into the symbol table. It should reference an undefined symbol table entry. The third field is the position, or offset, within the current segment at which relocation is to take place; it is printed in hexadecimal. The fourth field is the name of the segment referenced in the relocation: text, data, bss or EXT for external. The fifth field is the size of relocation: byte, word (2 bytes), or long. The last field will indicate, if present, that the relocation is relative.

If short form relocation is present, the format consist of three fields. The first field is the relocation command in hexadecimal. the second field contains the name of the segment referenced; text or data. The last field indicates the size of relocation: word or long.

Options and their meanings are:

-h
: Causes the executable binary file header and extended header to be printed out. Each field in the header or extended header is labeled. This is the default option.

-d
: Causes the data relocation records to be printed out.

-t Causes the text relocation records to be printed out.

-r
: Causes both text and data relocation to be printed.

-p
: Causes seek positions to be printed out as defined by macros in the include file, **<a.out.h>**.

-s
: Prints the symbol table.

-S
: Prints the file segment table with a header. (Only applicable to **x.out** segmented executable files.)

See Also

a.out(F), nm(CP)

Name

help – Asks for help about SCCS commands.

Syntax

help [args]

Description

help finds information to explain a message from an SCCS command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names. There are the following types of arguments:

type 1 Begins with nonnumerics, ends in numerics. The nonnumeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **ge6**, for message 6 from the *get* command).

type 2 Does not contain numerics (as a command, such as **get**)

type 3 Is all numeric (e.g., **212**)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try "help stuck".

Files

/usr/lib/help Directory containing files of message text

Name

ld – Invokes the link editor.

Syntax

ld [options] filename...

Description

ld is the XENIX link editor. It creates an executable program by combining one or more object files and copying the executable result to the file **a.ont**. The *filename* must name an object or library file. By convention these names have the ".o" (for object) or ".a" (for archive library) extensions. If more than one name is given, the names must be separated by one or more spaces. If errors occur while linking, *ld* displays an error message; the resulting **a.out** file is unexecutable.

ld concatenates the contents of the given object files in the order given in the command line. Library files in the command line are examined only if there are unresolved external references encountered from previous object files. Library files must be in *ranlib*(CP) format, that is, the first member must be named __.SYMDEF, which is a dictionary for the library. *ld* ignores the modification dates of the library and the __.SYMDEF entry, so if object files have been added to the library since __.SYMDEF was created, the link may result in an "invalid object module."

The library is searched iteratively to satisfy as many references as possible and only those routines that define unresolved external references are concatenated. Object and library files are processed at the point they are encountered in the argument list, so the order of files in the command line is important. In general, all object files should be given before library files. *ld* sets the entry point of the resulting program to the beginning of the first routine.

ld should be invoked using the *cc*(CP) instead of invoking it directly. *cc* invokes *ld* as the last step of compilation, providing all the necessary C-language support routines. Invoking *ld* directly is not recommended since failure to give command line arguments in the correct order can result in errors.

There are the following options:

-A *num*
Creates a standalone program whose expected load address (in hexadecimal) is *num*. This option sets the absolute flag in the header of the a.out file. Such program files can only be executed as standalone programs. Options **-A** and **-F** are mutually exclusive.

-B *num*
Sets the text selector bias to the specified hexadecimal number.

-c *num*
Alters the default target CPU in the *x.out* header. *num* can be 0, 1, 2, or 3 indicating 8086, 80186, 80286 and 80386 processors, respectively. The default on 8086/80286 systems is 0. The default on 80386 systems is 3. Note that this option only alters the default; if object modules containing code for a higher numbered processor are linked, then that will take precedence over the default.

-C
Causes the link editor to ignore the case of symbols.

-D *num*
Sets the data selector bias to the specified hexadecimal number.

-F *num*
Sets the size of the program stack to *num* bytes where num is a hexadecimal number. This option is ignored for 80386 programs which have a variable sized stack. By default 8086 programs have a variable stack located at the top of the first data segment, and 80286 programs have a fixed size 4096 byte stack. The **-F** option is incompatible with the **-A** option

-i
Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and shared by all users executing the file.

-m name
Creates a link map file named *name* that includes public symbols.

-M*x*
Specifies the memory model. *x* can have the following values:

s	small
m	middle
l	large
h	huge
e	mixed

-n *num*
Truncates symbols to the length specified by *num*.

-N *num*
Sets the pagesize to hex-*num* (which should be a multiple of 512) - the default is 1024 for 80386 programs. 8086/80186/80286 programs do not normally have page-aligned *x.out* files and the default for these is 0.

-o *name*
Sets the executable program filename to *name* instead of **a.out**.

-P
Disables packing of segments

-r Invokes the incremental linker, **/lib/ldr** , with the arguments passed to **ld** to produce a relocatable output file.

-R
Ensures that the relocation table is of non-zero size. Important for 8086 compatibility.

-Rd num
Specify the data segment relocation offset (80386 only). *num* is hexadecimal.

-Rt num
Specify the text segment relocation offset (80386 only) *num* is hexadecimal.

-s
Strips the symbol table.

-S *num*
Sets the maximum number of segments to *num*. If no argument is given, the default is 128.

-u *symbol*
Designates the specified *symbol* as undefined.

-v *num*
Specifies the XENIX version number. Acceptable values for *num* are 2, 3, or 5; 5 is the default.

Files

/bin/ld

See Also

ar(CP), masm(CP), cc(CP), ranlib(CP)

Notes

The user must make sure that the most recent library versions have been processed with *ranlib*(CP) before linking. If this is not done, *ld* cannot create executable programs using these libraries.

Name

lex – Generates programs for lexical analysis.

Syntax

lex [**−ctvn**] [file] ...

Description

lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx−z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *****, **+**, and **?** mean respectively; any nonnegative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character **.** is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character **$** at the end of an expression requires a trailing newline. The character **/** in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within **"** symbols or preceded by \\. Thus, **[a−zA−Z]+** matches a string of letters.

Three subroutincs defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless**(*p*) pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext+yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a nonblank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

Example

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switcb (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

The options must appear before any files. The option **-c** indicates C actions and is the default, **-t** causes the **lex.yy.c** program to be written instead to standard output, **-v** provides a one-line summary of statistics of the machine generated, **-n** will not print out the - summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n*
: number of positions is *n* (default 2000)

%n *n*
: number of states is *n* (500)

%t *n*
number of parse tree nodes is *n* (1000)

%a *n*
number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **−v** option, unless the **−n** option is used.

See Also

yacc(CP)
XENIX *Programmer's Guide*

Name

lint – Checks C language usage and syntax.

Syntax

lint [**−abchnpuvx**] [**−I***dir*] [**−DU***name*] [**−ol***lib*] [**−LARGE**] file ...

Description

lint attempts to detect features of the C program *file* that are likely to be bugs, nonportable, or wasteful. It also checks type usage more strictly than the C compiler. Among the things which are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

If more than one *file* is given, it is assumed that all the files are to be loaded together; they are checked for mutual compatibility. If routines from the standard library are called from *file*, *lint* checks the function definitions using the standard lint library **llibc.ln**. If *lint* is invoked with the **−p** option, it checks function definitions from the portable lint library **llibport.ln**.

Any number of *lint* options may be used, in any order. The following options are used to suppress certain kinds of complaints:

−a
Suppresses complaints about assignments of long values to variables that are not long.

−b
Suppresses complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in a large number of such complaints.)

−c
Suppresses complaints about casts that have questionable portability.

−h
Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

-u
Suppresses complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)

-v
Suppresses complaints about unused arguments in functions.

-x
Does not report variables referred to by external declarations but never used.

The following arguments alter *lint's* behavior:

-LARGE
Uses large model versions of the compiler and lint passes. This enables lint to handle flexnames (identifiers greater than 8 characters in length).

-n
Does not check compatibility against either the standard or the portable lint library.

-o
Creates a hashed (i.e. faster) version of lint library *lib* with suffix ".ln".

-p
Attempts to check portability to other dialects of C.

-llibname
Checks function definitions in the specified lint library. For example, **-lm** causes the library *llibm.ln* to be checked.

The **-D**, **-U**, and **-I** options of *cc*(CP) are also recognized as separate arguments.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
At appropriate points stops comments about unreachable code.

/*VARARGS*n**/
Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
Turns on the **-v** option for the next function.

/*LINTLIBRARY*/
Shuts off complaints about unused functions in this file.

lint produces its first output on a per source file basis. Complaints regarding included files are collected and displayed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename is displayed followed by a question mark.

Files

/usr/lib/lint[12] Program files

/usr/lib/llibc.ln, /usr/lib/llibport.ln, /usr/lib/llibm.ln, /usr/lib/llibdbm.ln, /usr/lib/llibtermlib.ln
Standard lint libraries (binary format)

/usr/lib/llibc, /usr/lib/llibport, /usr/lib/llibm, /usr/lib/llibdbm, /usr/lib/llibtermlib
Standard lint libraries (source format)

/usr/tmp/*lint* Temporaries

See Also

cc(CP)

Notes

exit(S), and other functions which do not return, are not understood. This can cause improper error messages.

Name

lorder – Finds ordering relation for an object library.

Syntax

lorder file ...

Description

lorder creates an ordered listing of object filenames, showing which files depend on variables declared in other files. The *file* is one or more object or library archive files (see *ar*(CP)). The standard output is a list of pairs of object filenames. The first file of the pair refers to external identifiers defined in the seeond. The output may be processed by *tsort*(CP) to find an ordering of a library suitable for one-pass access by *ld*(CP).

Example

The following command builds a new library from existing **.o** files:

```
ar cr library `lorder *.o |tsort`
```

Files

*symref, *symdef   Temp files

See Also

ar(CP), ld(CP), tsort(CP)

Notes

Object files whose names do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

Name

m4 – Invokes a macro processor.

Syntax

m4 [options] [files]

Description

m4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument *files* is processed in order; if there are no files, or if a filename is **–**, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e
: Operates interactively. Interrupts are ignored and the output is unbuffered.

–s
: Enables line sync output for the C preprocessor (#line ...)

–B*int*
: Changes the size of the push-back and argument collection buffers from the default of 4,096.

–H*int*
: Changes the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int*
: Changes the size of the call stack from the default of 100 slots. Macros take three slots, and nonmacro arguments take one.

–T*int*
: Changes the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any filenames and before any **–D** or **–U** flags:

–D*name*[=*val*]
: Defines *name* to *val* or to null in *val*'s absence.

–U*name*
: Undefines *name*.

Macro Calls

Macro calls have the form:

name(arg1,arg2, . . ., argn)

The (must immediately follow the name of the macro. If a defined macro name is not followed by a (, it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Left and right single quotation marks are used to quote strings. The value of a quoted string is the string stripped of the quotation marks.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

m4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define — The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; $# is replaced by the number of arguments; $* is replaced by a list of all the arguments separated by commas; $@ is like $*, but each argument is quoted (with the current quotation marks).

undefine — Removes the definition of the macro named in its argument.

defn — Returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef — Like *define*, but saves any previous definition.

popdef — Removes current definition of its argument(s), exposing the previous one if any.

ifdef
: If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word **XENIX** is predefined in *m4*.

shift
: Returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote
: Changes quotation marks to the first and second arguments. The symbols may be up to five characters long. *changequote* without arguments restores the original values (i.e., `').

changecom
: Changes left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert
: *m4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert
: Causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum
: Returns the value of the current output stream.

dnl
: Reads and discards characters up to and including the next newline.

ifelse
: Has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or if it is not present, null.

incr
: Returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr	Returns the value of its argument decremented by 1.
eval	Evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, %, ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.
len	Returns the number of characters in its argument.
index	Returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.
substr	Returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.
translit	Transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.
include	Returns the contents of the file named in the argument.
sinclude	Identical to *include*, except that it says nothing if the file is inaccessible.
syscmd	Executes the XENIX command given in the first argument. No value is returned.
sysval	Is the return code from the last call to *syscmd*.
maketemp	Fills in a string of XXXXX in its argument with the current process ID.
m4exit	Causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.
m4wrap	Argument 1 will be pushed back at final EOF; example: m4wrap(\`cleanup()´)
errprint	Prints its argument on the diagnostic output file.
dumpdef	Prints current names and definitions, for the named items, or for all if no arguments are given.

traceon	With no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.
traceoff	Turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

Name

make – Maintains, updates, and regenerates groups of programs.

Syntax

make [**−f** makefile] [**−p**] [**−i**] [**−k**] [**−s**] [**−r**] [**−n**] [**−b**] [**−e**] [**−t**] [**−q**] [**−d**] [names]

Description

The following is a brief description of all options and some special names:

−f *makefile* Description filename. *makefile* is assumed to be the name of a description file. A filename of **−** denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

−p Prints out the complete set of macro definitions and target descriptions.

−i Ignores error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

−k Abandons work on the current entry, but eontinues on other branches that do not depend on that entry.

−s Silent mode. Does not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

−r Does not use the built-in rules.

−n No execute mode. Prints commands, but does not execute them. Even lines beginning with an @ are printed.

−b Compatibility mode for old makefiles.

−e Environment variables override assignments within makefiles.

−t Touches the target files (causing them to be up-to-date) rather than issues the usual commands.

−d Debug mode. Prints out detailed information on files and times examined.

−q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up-to-date.

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the **−s** option.

.IGNORE Same effect as the **−i** option.

make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **−f** option is present, *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* are tried in order. If *makefile* is **−**, the standard input is taken. More than one **−f** makefile argument pair may appear.

make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, nonnull list of targets, then a **:**, then a (possibly null) list of prerequisite files or dependencies. Text following a **;** and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><newline> sequence. (#) and newline surround comments.

The following *makefile* says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**:

```
pgm: a.o b.o
    cc a.o b.o -o pgm
a.o: incl.h a.c
    cc -c a.c
b.o: incl.h b.c
    cc -c b.c
```

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the **−s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the first character of the command is @. The **−n** option specifies printing without execution; however, if the command line has the string

$(MAKE) in it, the line is always executed (see discussion of the MAKEFLAGS macro under *Environment*). The −t (touch) option updates the modified date of a file without executing any commands.

Commands returning nonzero status normally terminate *make*. If the −i option is present, or the entry *.IGNORE:* appears in *makefile*, or if the line specifying the command begins with <tab><hyphen>, the error is ignored. If the **−k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **−b** option allows old makefiles (those written for the old version of **make**) to run without errors. The difference between the old version of **make** and this version is that this version requires all dependency lines to have a (possibly null) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless *.PRECIOUS* is on it.

Environment

The environment is read by **make**. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The −e option causes the environment to override the macro assignments in a makefile.

The *MAKEFLAGS* environment variable is processed by **make** as containing any legal input option (except *−f*, *−p*, and *−d*) defined for the command line. Further, upon invocation, **make** "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, *MAKEFLAGS* always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the *−n* option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a *make −n* recursively on a whole software system to see what would have been executed. This is because the *−n* is put in *MAKEFLAGS* and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional

:subst1=subst2 is a substitute sequence. If it is specified, all nonoverlapping occurrences of **subst1** in the named macro are replaced by **subst2**. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets:

*$** The macro *$** stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@* The *$@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<* The *$< macro is only evaluated for inference rules or the *.DEFAULT* rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent filename). Thus, in the *.c.o* rule, the *$<* macro would evaluate to the *.c* file. An example for making optimized *.o* files from *.c* files is:

```
.c.o:
    cc -c -O $*.c
```

or:

```
.c.o:
    cc -c -O $<
```

$?* The *$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

$% The *$%* macro is only evaluated when the target is an archive library member of the form *lib(file.o)*. In this case, *$@* evaluates to *lib* and *$%* evaluates to the library member, *file.o*.

Four of the five macros can have alternative forms. When an upper case *D* or *F* is appended to any of the four macros the meaning is changed to "directory part" for *D* and "file part" for *F*. Thus, *$(@D)* refers to the directory part of the string *$@*. If there is no directory part *./* is generated. The only macro excluded from this alternative form is *$?*.

Suffixes

Certain names (for instance, those ending with *.o*) have default dependents such as *.c*, *.s*, etc. If no update commands for such a file appear in *makefile*, and if a default dependent exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

The internal rules for *make* are contained in the source file *rules.c* for the *make* program. These rules can be locally modified. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the **(null)** string which *printf*(S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(F)). Thus, the rule *.c~.o* would transform an SCCS C source file into an object file (*.o*). Because the *s.* of the SCCS files is a prefix it is incompatible with *make*'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e. *.c:*) is the definition of how to build **x** from **x**.*c*. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for *.SUFFIXES*. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

The default list is:

```
.SUFFIXES: .o .c .y .l .s
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; *.SUFFIXES:* with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
    cc a.o b.o -o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(CP), *lex*(CP), and *yacc*(CP) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with *.c.o:* as the target and no dependents. Shell commands associated with the target define the rule for making a *.o* file from a *.c* file. Any target that has no slashes in it and starts with a dot is identified as a rule and not as a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus *lib(file.o)* and *$(LIB)(file.o)* both refer to an archive library which contains *file.o*. (This assumes the **LIB** macro has been previously defined.) The expression *$(LIB)(file1.o file2.o)* is not legal. Rules pertaining to archive libraries have the form *.XX*.a where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have *lib(file.o)* depend upon *file.o* explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:
    lib(file1.o) lib(file2.o) lib(file3.o)
    @echo lib is now up to date
.c.a:
    $(CC) -c $(CFLAGS) $<
    ar rv $@ $*.o
    rm -f $*.o
```

In fact, the *.c.a* rule listed above is built into *make* and is unnccessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:
    lib(file1.o) lib(file2.o) lib(file3.o)
    $(CC) -c $(CFLAGS) $(?:.o=.c)
    ar rv lib $?
    rm $?  @echo lib is now up to date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The *$?* list is defined to be the set of object filenames (inside *lib*) whose C source files are out of date. The substitution mode translates the *.o* to *.c*. (Unfortunately, one cannot as yet transform to *.c~*) Note also, the disabling of the *.c.a:* rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

Files

[Mm]akefile

s.[Mm]akefile

See Also

sh(C)

Notes

Some commands return nonzero status inappropriately; use **-i** to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(C), are ineffectual across newlines in *make*. The syntax *(lib(file1.o file2.o file3.o)* is illegal. You cannot build *lib(file.o)* from *file.o*. The macro *$(a:.o=.c~)* is not available.

Name

masm – Invokes the XENIX assembler.

Syntax

masm [*options*] *sourcefile*

Description

masm is the XENIX 8086/286/386 assembler. It reads and assembles 8086/80286/80386 assembly language instructions from the source file named *sourcefile*. It then creates a linkable object file name *sourcefile*.o, or an executable program named **a.out**.

The extension .s is recommended but not required. If this extension is not given, *masm* displays a warning and continues processing.

There are the following options:

-a
This options puts the assembled output segments in alphabetic order before copying them to the object file.

-c
Outputs cross reference data for each assembled file to *filename.crf*.

-C
Outputs cross reference data for a set of assembled file. The cross reference data is written to files with the same names as the input files, with the filename extension ".crf."

-d
Adds a pass 1 listing to the assembly listing file *filename.lst*.

-Dsym
Defines the symbol appended to the **-D** flag as a null TEXTMACRO.

-e
Generates floating point code to emulate the 8087 or 287 coprocessor. Programs created with this option must be linked with an appropriate math library before being executed.

-Ipath
Defines the path appended to the **-I** flag as the search path for include files. Up to 10 include paths are allowed in one invocation of *masm*.

- l[*listfile*]
  Creates an assembly listing file with the same basename as the *sourcefile* or, if the *listfile* parameter is given, with that name but with a ".lst" extension. The file lists the source instructions, the assembled (binary code) for each instruction and any assembly errors. If filename is "-," the listing is written to *stdout*.

- **Mx**
  This option directs *masm* to preserve lower case letters in public and external names only when copying these names to the object file. For all other purposes, *masm* converts the lower case to upper case.

- **Mu**
  Disables case sensitivity. Upper case is now treated as identical to lower case.

- **Ml**
  Leave case of symbols alone.

- **n**
  This option generates information about the symbols used in the assembled programs. The -l option must also be used for this option to take effect.

- **o***objfile*
  Copies the assembled instructions in octal to the file named *objfile*. This file is executable only if no errors occurred during the assembly. This option overrides the default object file name.

- **O***objfile*
  Copies the assembled instructions in binary to the file named *objfile*.

- **r**
  Generates floating point code that can only be executed by an 8087 or 287 coprocessor.

- **v**
  Prints verbose error statistics on console. If not selected, only error counts are displayed.

- **x**
  displays error messages on the standard error channel, in addition to the messages generated in the listing file.

- **X**
  Copies to the assembly listing all statements forming thef body of an IF directive whose expression (or condition) evaluates to false.

Files

/bin/masm

See Also

a.out(F), cc(CP), ld(CP)
Macro Assembler User's Guide

Notes

The default options are **-Ml** and **-e** which enable case sensitivity and allow emulation of a floating point processor. The options are flags with the following default settings:

Flag	**Default**	**Meaning of TRUE condition**
a	FALSE	Outputs segments alphabetically
c	FALSE	Outputs cross reference data
C	FALSE	Outputs cross reference data
d	FALSE	Adds pass 1 listing to filename.lst
Dsym	NULL	No meaning if not defined
e	FALSE	Floating Point emulation
I path	NULL	No meaning if not defined
llistfile	sourcefile.lst	Sourcefile is the default filename
M	l	Leave symbol case alone
n	TRUE	Outputs symbols if -l selected
o	TRUE	Assembled output in binary
O	FALSE	Assembled output in octal
r	TRUE	Real 8087 instead of emulated format
v	FALSE	Prints verbose error statistics
x	TRUE	Displays errors on console
X	FALSE	Toggle setting of conditional flag

Return Value

The *masm* exit codes have the following meanings:

Code	**Meaning**
0	No error
1	Argument error
2	Unable to open input file
3	Unable to open listing file
4	Unable to open object file
5	Unable to open cross reference file
6	Unable to open include file
7	Assembly errors. If fatal, the object file is deleted.

MASM (CP)

8	Memory allocation error
9	Real number input not allowed in this version.

Name

mkstr – Creates an error message file from C source.

Syntax

mkstr [–] messagefile prefix file ...

Description

mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

mkstr will process each specified *file*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. The optional dash (–) causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr* ed program.

A typical *mkstr* command line is

```
mkstr pistrings xx *.c
```

This command causes all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file, *mkstr* keys on the string 'error("' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a newline character; the null character terminates the message so it can be easily used when retrieved, the newline character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message. For example, the command changes

```
error("Error on reading", a2, a3, a4);
```

into

error(*m*, a2, a3, a4);

where *m* is the seek position of the string in the resulting error message file. The programmer must create a routine *error* which opens the message file, reads the string, and prints it out. The following example illustrates such a routine.

Example

```
char    efilname[] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
int a1, a2, a3, a4;
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0) {
                printf("Unable to find error msg at seek address %d0,a1);
                exit(1);
                }
        printf(buf, a2, a3, a4);
}
```

See Also

lseek(S), xstr(CP)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

All the arguments except the name of the file to be processed are unnecessary.

Name

nm – Prints name list.

Syntax

nm [**−acgnoOprsSuv**] [+offset] [file ...]

Description

nm prints the name list (symbol table) of each object *file* in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no *file* is given, the symbols in **a.out** are listed.

Each symbol name is preceded by its value in hexadecimal (blanks if undefined) and one of the letters U (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), S (segment name), **C** (common symbol), **K** (8086 common segment), or S (segment name). If the symbol table is in segmented format, symbol values are displayed as **segment:offset**. If the symbol is local (non-external), the type letter is in lowercase. The output is sorted alphabetically.

Options are:

−a Attempt to print the namelist of all modules in an archive library. Normally, *nm* silently ignores any library members which are not valid object modules. Using this option causes *nm* to report an error for all such modules. Note that the first member in any library which has been processed by *ranlib*(CP) is called ____.SYMDEF and is not a valid object module, thus the **−a** option will always produce at least one error message when used on such a library.

−c Print only C program symbols (symbols which begin with '_') as they appeared in the C program.

−g Print only global (external) symbols.

−n Sort numerically rather than alphabetically.

−o Prepend file or archive element name to each output line rather than only once.

−O Print symbol values in octal.

−p Don't sort; print in symbol-table order.

−r Sort in reverse order.

−s Sort by size of symbol and display each symbol's size instead of value. The last symbol in each text or data segment may be assigned a size of 0. This implies the **−n** option.

−S Switch the display format. If the symbol table is in segmented format, print values in non-segmented format. If not segmented, print values in segmented format. Segment offsets in 386 object modules and executable files are 32 bits rather than 16 bits.

−u Print only undefined symbols.

−v Also describe the object file and symbol table format.

Files

a.out Default input file

See Also

ar(CP), ar(F), a.out(F)

Name

prof – Displays profile data.

Syntax

prof [**-a**] [**-l**] [file]

Description

prof interprets the file **mon.out** produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file (**a.out** default) is read and correlated with the **mon.out** profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **-a** option is used, all symbols are reported rather than just external symbols. If the **-l** option is used, the output is listed by symbol value rather than decreasing percentage.

To cause calls to a routine to be tallied, the **-p** option of *cc* **must** have been given when the file containing the routine was compiled. This option also arranges for the **mon.out** file to be produced automatically.

Files

mon.out For profile

a.out For namelist

See Also

monitor(S), profil(S), cc(CP)

Notes

Beware of quantization errors.

If you use an explicit call to *monitor*(S) you will need to make sure that the buffer size is equal to or smaller than the program size.

Warning

Profiling gives incorrect results for hybrid model 286 programs (i.e. those with 16 bit text pointers within modules and 32 bit text pointers between modules).

Name

prs – Prints an SCCS file.

Syntax

prs [**−d**[dataspec]] [**−r**[SID]] [**−e**] [**−l**] [**−a**] files

Description

prs prints, on the standard output, all or part of an SCCS file (see *sccsfile*(F)) in a user supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with **s.**), and unreadable files are silently ignored. If a name of **−** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; nonSCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of options, and filenames.

All the described options apply independently to each named file:

−d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *Data Keywords*) interspersed with optional user-supplied text.

−r[*SID*] Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

−e Requests information for all deltas created *earlier* than and including the delta designated via the **−r** option.

−l Requests information for all deltas created *later* than and including the delta designated via the **−r** option.

−a Requests printing of information for both removed, i.e., delta type = *R*, (see *rmdel*(CP)) and existing, i.e., delta type = *D*, deltas. If the **−a** option is not specified, information for existing deltas only is provided.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(F)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of the user-supplied text and appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either simple, in which keyword substitution is direct, or multiline, in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords. A tab is specified by \t and carriage return/newline is specified by \n.

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer wh created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module names	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:R:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(C) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(C) string	N/A	:Z::Y::M::I::Z:	S
:Z:	*what*(C) string delimiter	N/A	@(#)	S
:F:	SCCS filename	N/A	text	S
:PN:	SCCS file pathname	N/A	text	S

* :Dt: = :DT::I::D::T::P::DS::DP:

Examples

The following:

```
prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r
s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only option allowed to be used with the *special case* is the **-a** option.

Files

/tmp/pr?????

See Also

admin(CP), delta(CP), get(CP), help(CP), sccsfile(F)

Diagnostics

Use *help* (CP) for explanations.

Name

ranlib – Converts archives to random libraries.

Syntax

ranlib archive...

Description

ranlib converts each *archive* to a form which can be loaded more rapidly by the loader, by adding a table of contents named **__.SYMDEF** to the beginning of the archive. It uses *ar*(CP) to reconstruct the archive, so sufficient temporary file space must be available in the file system containing the current directory.

See Also

ld(CP), ar(CP), copy(C), settime(C)

Notes

Failure to process a library with *ranlib*, or failure to reprocess a library with *ranlib*, will cause *ld* to fail. Because generation of a library by *ar* and randomization by *ranlib* are separate, phase errors are possible. The loader *ld* warns when the modification date of a library is more recent than the creation of its dictionary; but this means you get the warning even if you only copy the library.

Name

ratfor – Converts Rational FORTRAN into standard FORTRAN.

Syntax

ratfor [option ...] [filename ...]

Description

ratfor converts a rational dialect of FORTRAN into ordinary irrational FORTRAN. *ratfor* provides control flow constructs essentially identical to those in C:

```
statement grouping:
    { statement; statement; statement }

decision-making:
    if (condition) statement [ else statement ]
    switch (integer value) {
            case integer:    statement
            ...
            [ default: ]     statement
    }

loops:
    while (condition) statement
    for (expression; condition; expression) statement
    do limits statement
    repeat statement [ until (condition) ]
    break [n]
    next [n]
```

It also provides some additional syntax to make programs easier to read and write:

```
Free form input:
    multiple statements/line; automatic continuation

Comments:
    # this is a comment

Translation of relationals:
    >, >=, etc., become .GT., .GE., etc.

Return (expression)
    returns expression to caller from function
```

Define:
    define name replacement

Include:
    include filename

The following options are available:

−h Causes quoted strings to be turned into 27H constructs.

−C Copies comments to the output, and attempts to format it neatly. Normally, continuation lines are marked with an & in column 1.

−6x Makes the continuation character **x** and places it in column 6.

Name

regcmp – Compiles regular expressions.

Syntax

regcmp [**–**] files

Description

regcmp, in most cases, precludes the need for calling *regcmp* (see *regex*(S)) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in **file** and places the output in **file.i**. If the **–** option is used, the output will be placed in **file.c**. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotation marks. The output of *regcmp* is C source code. Compiled regular expressions are represented as **extern char** vectors. *File*.**i** files may thus be *included* into C programs, or *file*.**c** files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) applies the regular expression named *abc* to *line*. Diagnostics are self-explanatory.

Examples

```
name    "([A-Za-z][A-Za-z0-9_]*)$0"

telno   "\({0,1}([2-9][01][1-9])$0\){0,1} *"
        "([2-9][0-9]{2})$1[ -]{0,1}"
        "([0-9]{4})$2"
```

In the C program that uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line*.

See Also

regex(S)

Name

rmdel – Removes a delta from an SCCS file.

Syntax

rmdel –rSID files

Description

rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must *not* be that of a version being edited for the purpose of making a delta. That is, if a *p-file* exists for the named SCCS file, the SID specified must *not* appear in any entry of the *p-file*(see *get*(CP)).

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

Files

x-file See *delta*(CP)

z-file See *delta*(CP)

See Also

delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Name

sact – Prints current SCCS file editing activity.

Syntax

sact files

Description

sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(CP) with the **−e** option has been previously executed without a subsequent execution of *delta*(CP). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that nonSCCS files and unreadable files are silently ignored. If a name of − is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1	Specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta
Field 2	Specifies the SID for the new delta to be created
Field 3	Contains the logname of the user who will make the delta i.e., executed a *get* for editing
Field 4	Contains the date that **get −e** was executed
Field 5	Contains the time that **get −e** was executed

See Also

delta(CP), get(CP), unget(CP)

Diagnostics

Use *help*(CP) for explanations.

Name

sccsdiff – Compares two versions of an SCCS file.

Syntax

sccsdiff −rSID1 **−r**SID2 [**−p**] [**−sn**] files

Description

sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

−r*SID?* *SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(C) in the order given.

−p Pipe output for each file through *pr*(C).

−s*n* *n* is the file segment size that *bdiff* will pass to *diff*(C). This is useful when *diff* fails due to a high system load.

Files

/tmp/get????? Temporary files

See Also

bdiff(C), get(CP), help(CP), pr(C)

Diagnostics

file: *No differences* If the two versions are the same.

Use *help*(CP) for explanations.

Name

sdb – Invokes symbolic debugger.

Syntax

sdb [*objfil* [*corfil* [*directory:directory*]]]

Description

sdb is a symbolic debugger which can be used with C programs.

Objfil is an executable program file which has been compiled with the **–Zi** (debug) option and linked with the **–I** option. The default for *objfil* is **a.out**. *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**. A "–" in place of *corfil* forces *sdb* to ignore any core image file. The colon separated directory list is used to locate the source files used to build objfil.

It is useful to know that at any time there is a *current line* and *current file*. They are initially set to the first line in **main**(). The current line and file may be changed with the source file examination commands.

Names of variables are written just as they are in C programs. Variables local to a procedure may be accessed using the form *procedure.variable*. If no procedure name is given, the procedure containing the current line is used by default.

You can also refer to structure members as *variable.member*, pointers to structure members as *variable–>member* and array elements as *variable*[*number*]. Pointers may be de-referenced by using the form *pointer*[0]. You can also use combinations of these forms.

It is also possible to specify a variable by its address. You can use all forms of integer constants which are valid in C programs, so that addresses and numbers may be input in decimal, octal, or hexadecimal.

Line numbers in source programs are referred to as *filename:number* or *procedure:number*. In either case the number is relative to the beginning of the file. If no procedure or filename is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

There are several kinds of commands available to the *sdb* debugger as described in the following sections. *sdb* commands appear in boldface type. For all commands, items in brackets ([ ]) are optional.

Data Examination Commands

t Displays a stack trace.

T Prints the top line of the stack trace.

variable/[*clm*]

Displays the value of *variable* according to length *l* and format *m*. A numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. If *l* and *m* are omitted, *sdb* chooses a format suitable for the variable type as declared in the program. The length specifiers are:

b One byte

h Two bytes (half word)

l Four bytes (long word)

Legal values for *m* are:

- **c** Character
- **d** Decimal
- **u** Unsigned decimal
- **o** Octal
- **x** Hexadecimal
- **f** 32 bit single precision floating point
- **g** 64 bit single precision floating point
- **s** Assumes *variable* is a string pointer and prints characters starting at the address pointed to by the variable.
- **a** Prints characters starting at the variable's address.
- **i** Disassembles with numeric/symbolic addresses.

The length specifiers are only effective with the formats **c**, **d**, **u**, **o**, and **x**. If one of these formats is specified and *l* is omitted, the length defaults to two bytes. If a numeric length specifier is used for the format variable then that many characters are printed. Otherwise, successive characters are printed until either a null byte is reached or 128 characters are printed.

linenumber?[*clm*]

Prints the value at the address from **a.out** or **i** space given by *linenumber*, according to the format *lm*. The default format is **i**.

variable=[*lm*]
linenumber=[*lm*]
number=[*lm*]

Prints the address of *variable* or *linenumber* in the format specified by *lm* . If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal, and hexadecimal. A single number cannot be used as a line number because the command would be ambiguous; the *proc*:*number* form must be used.

variable!*value*

Sets variable to the given value. The value may be any valid C expression.

x Displays the machine registers and current machine-language instruction.

X Displays the current machine-language instruction.

Source File Examination Commands

e Displays current procedure and filenames.

e *procedure*

Sets the current file and current line to the file containing *procedure*.

e *filename*

Sets the current file and current line number to the first line in *filename* .

/*regular expression*[/]

Searches forward from the current line for a line containing a string matching *regular expression* as in *ed*(C).

?*regular expression*[?]

Searches backward from the current line for a line containing a string matching *regular expression* as in *ed*(C).

p Prints the current line.

z Prints the current line followed by the next nine lines. Sets the current line to the last line printed.

w Creates a window by printing ten lines around the current line.

number

Sets the current line to the given line number and displays the line.

[*count*]+

Advances the current line by *count* lines and display the new line. If *count* is omitted, the default is one line.

[*count*]-

Retreats from the current line by count lines and display the new line. If *count* is omitted, the default is one line.

Execution Control Commands

L Load the program to be debugged but do not run it. If you wish to examine the initial values of memory locations before the program has started to run, or if you wish to disassemble portions of the program without actually running it, you must first enter the L command.

[*count*] **r** [*args*]
[*count*] **R**
Runs the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

[*linenumber*] **c** [*count*]
[*linenumber*] **C** [*count*]
Continues after a breakpoint or interrupt. If *count* is given, it specifies the number of breakpoints to be ignored. **C** continues with the signal which caused the program to stop reactivated and **c** ignores it. If a line number is specified then a

temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes.

linenumber **g** [*count*]
: Continues after a breakpoint with execution resumed at the given line. If *count* is given, it specifies the number of breakpoints to be ignored.

[*count*] **s**
: Single steps. Runs the program through *count* lines. If no count is given then the program is run for one line.

[*count*] **S**
: Single steps but steps through subroutine calls.

[*count*] **i**
: Machine-language single steps. Runs the program through *count* machine-language instructions. If no count is given then one machine-language instruction is executed.

[*count*] **I**
: Machine-language single steps, but steps through call instructions.

variable$**m** [*count*]
: Single steps (as with **s**) until the specified location is modified with a new value. *Count* specifies the number of instructions to step; if omitted, *count* is effectively infinity. The variable must be accessible from the current procedure. Since this command is performed by software, it can be very slow.

[*level*] **v**
: Switches verbose mode on and off, for use with single stepping with **S**, **s**, or **m**. If *level* is omitted or is zero, then just the current source file and/or subroutine name is printed when either changes. If *level* is one, each C source line is printed before it is executed; if *level* is two, each assembler line statement is also printed. The **v** command turns verbose mode off if it is on for any level.

k Kills the debugged program.

procedure(*arg1*,*arg2*,...)
procedure(*arg1*,*arg2*,...)/*m*
: Executes the named procedure with the given arguments. The second form causes the value to be returned by the procedure to be printed according to format *m*. If no format is given, it defaults to **d**.

[*linenumber*] **b** [*commands*]
Sets a breakpoint at the given line. If a procedure name without a line number is given (e.g., "main"), a breakpoint is placed at the first line in the procedure. If no *linenumber* is given, a breakpoint is placed at the current line. If no *commands* are given then execution stops just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons.

B Prints a list of the currently active breakpoints.

[*linenumber*] **d**
Deletes a breakpoint at the given line. If no *linenumber* is given, then the breakpoints are deleted interactively: each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d**, then the breakpoint is deleted.

D Deletes all breakpoints.

l Prints the last executed line. Makes the last executed line the current line.

linenumber **a**
Announces. If *linenumber* is of the form *proc:number* or *number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc:*, the command effectively does a *proc:* **b T**.

Miscellaneous Commands

!*command*
Interprets command. Command interpreter executes *command.*

newline
Advances the current line by one line and prints the new current line if the previous command printed a source line. Displays the next memory location if the previous command displayed a memory location.

Ctrl-D
Scrolls. Prints the next ten lines of instructions, source or data depending on which was printed last.

< filename
Reads commands from filename until the end of file is reached, and then continues to accept commands from standard input. When *sdb* is told to display a variable by a

command in such a file, the variable name is displayed along with the value. This command may not be nested; the redirection character (<) may not appear as a command in a file.

"*string*
Prints the given string. The C escape sequences of the form *character* are recognized, where *character* is a non-numeric character.

q Exits the debugger.

Debugger Commands

V Prints the version number.

Q Prints a list of procedures and files being debugged.

Files

a.out
core

See Also

adb(CP), a.out(F), cc(CP), core(F), ld(CP)

Notes

In order to make use of the symbolic debugging features of *sdb*, the program being debugged must have been compiled with the **−Zi** option. *sdb* does not use the ordinary symbol table information in an *a.out* file and has limited facilities for debugging at the machine code level. If you have to debug a program that has been compiled without using the **−Zi** option, it may be preferable to use *adb*.

Name

size – Prints the size of an object file.

Syntax

size [object ...]

Description

size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in decimal and hexadecimal, of each object-file argument. If no file is specified, **a.out** is used.

See Also

a.out(F)

Name

spline – Interpolates smooth curve.

Syntax

spline [option] ...

Description

spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output has two continuous derivatives, and enough points to look smooth when plotted.

The following options are recognized, each as a separate argument.

–a Supplies abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

–k The constant k used in the boundary value computation

$$y_0'' = ky_1', \ldots, y_n'' = ky_{n-1}'$$

is set by the next argument. By default $k = 0$.

–n Spaces output points so that approximately n intervals occur between the lower and upper x limits. (Default $n = 100$.)

–p Makes output periodic, i.e. matches derivatives at ends. First and last input values should normally agree.

–x Next 1 (or 2) arguments are lower (and upper) x limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

Diagnostics

When data is not strictly monotone in x, *spline* reproduces the input without interpolating extra points.

Notes

A limit of 1000 input points is silently enforced.

Name

stackuse – Determines stack requirements for C programs.

Syntax

stackuse [**-m** startsym] [**-r** fakeref] [**-s** libstack] [**-a**] file ...

Description

stackuse determines the stack requirements of one or more C language programs. It displays the name of the *main* routine in a file, its stack requirements in bytes, and the number of recursive routines. All command line switches are optional.

-m*startsym* Prints only the specified start ("main") symbol. If this option is not specified all start symbols (those which are not called by anybody) will be printed.

-r*fakeref* Uses the named file *fakeref* as a fake references file. The format is: *parent child* . The special *parent* .LEAF is a meta-parent meaning all leaf nodes.

-s*libstack* Uses the named file as library of costs for external routines. The format is: *subr stack*. The special *subr* .UNDEF is a meta-subroutine meaning all undefined routines.

-a Prints data for all symbols, not just start symbols.

The **-r** and **-s** options may be repeated an arbitrary number of times. The effect is additive rather than destructive. In the case of duplicate definitions, the first is used.

Lines of the **-r** and **-s** files which begin with a pound sign (#) are treated as comments and otherwise are ignored.

Files

/usr/lib/stackuse/*	Passes, libraries
/tmp/*	Temporaries used by passes.

Diagnostics

Usage (fatal).

Redefinitions in **-r**, **-s** files, or in the source (warning).

Presence of routines for which no stack value is provided (warning).

Notes

For the **libstack** and **fakeref** files, a comment character (#) is used.

Name

strings – Finds the printable strings in an object file.

Syntax

strings [–] [–o] [–*number*] file ...

Description

strings looks for ASCII strings in a binary file. A string is any sequence of four or more printing characters ending with a newline or a null character. Unless the – flag is given, *strings* only looks in the initialized data space of object files. If the **–o** flag is given, then each string is preceded by its decimal offset in the file. If the *–number* flag is given then *number* is used as the minimum string length rather than 4.

strings is useful for identifying random object files and many other things.

See Also

hd(C), od(C)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Name

strip – Removes symbols and relocation bits.

Syntax

strip [-MNSdehrstx] file ...

Description

strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and link editor. This is useful for saving space after a program has been debugged.

If *name* is an archive file, *strip* will remove the local symbols from any *a.out* format files it finds in the archive. Certain libraries, such as those residing in **/lib**, have no need for local symbols. By deleting them, the size of the archive is decreased and link editing performance is increased.

There are several options for use with *strip*:

-M	Strip all memory image segments.
-N	Strip all non-memory image segments.
-S	Strip the segment table only.
-h	Strip header and extended header.
-e	Strip extended header.
-d	Strip data and data relocation.
-t	Strip text and text relocation.
-r	Strip all relocation except x.out's "short form."
-x	Strip all relocation.
-s	Strip symbol table.

The effect of *strip* is the same as use of the -s option of *ld*.

Files

/tmp/stm* Temporary file

See Also

ld(C)

Name

time – Times a command.

Syntax

time command

Description

The given *command* is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on the standard error.

See Also

times(S)

Name

tsort – Sorts a file topologically.

Syntax

tsort [file]

Description

tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

See Also

lorder(CP)

Diagnostics

Odd data: There is an odd number of fields in the input file.

Notes

The *sort* algorithm is quadratic, which can be slow if you have a large input list.

Name

unget – Undoes a previous get of an SCCS file.

Syntax

unget [**−r**SID] [**−s**] [**−n**] files

Description

unget undoes the effect of a **get −e** done prior to creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified as a named file, except that nonSCCS files and unreadable files are silently ignored. If a name of **−** is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Options apply independently to each named file.

−r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta".) The use of this option is necessary only if two or more versions of the same SCCS file have been retrieved for editing by the same person (login name). A diagnostic results if the specified *SID* is uncertain, or if it is necessary and omitted on the command line.

−s Suppresses the printout, on the standard output, of the intended delta's *SID*.

−n Causes the retention of the file which would normally be removed from the current directory.

See Also

delta(CP), get(CP), sact(CP)

Diagnostics

Use *help*(CP) for explanations.

Name

val – Validates an SCCS file.

Syntax

val –

val [–**s**] [–**r**SID] [–**m**name] [–**y**type] files

Description

val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of options, which begin with a –, and named files.

val has a special argument, –, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option apply independently to each named file on the command line:

–s The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

–r*SID* The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if tbe *SID* is ambiguous (e. g., **r1** is ambiguous because it physically does not exist but implics 1.1, 1.2, etc. which may exist) or invalid (e. g., **r1.0** or **r1.1.0** are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

–m*name* The argument value *name* is compared with the SCCS %M% keyword in *file*.

–y*type* The argument value *type* is compared with tbe SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = Missing file argument

bit 1 = Unknown or duplicate option

bit 2 = Corrupted SCCS file

bit 3 = Can't open file or file not SCCS

bit 4 = *SID* is invalid or ambiguous

bit 5 = *SID* does not exist

bit 6 = %Y%, **-y** mismatch

bit 7 = %M%, **-m** mismatch

Note that *val* can process two or more files on a given command line and in turn can process multiple command line (when reading the standard input). In these cases an aggregate code is returned; a logical OR of the codes generated for each command line and file processed.

See Also

admin(CP), delta(CP), get(CP), prs(CP)

Diagnostics

Use *help*(CP) for explanations.

Notes

val can process up to 50 files on a single command line.

Name

xref – Cross-references C programs.

Syntax

xref [file ...]

Description

xref reads the named *files* or the standard input if no file is specified and prints a cross reference consisting of lines of the form

identifier filename line numbers ...

Function definition is indicated by a plus sign (**+**) preceding the line number.

See Also

cref(CP)

Name

xstr – Extracts strings from C programs.

Syntax

xstr [**–c**] [**–**] [file]

Description

xstr maintains a file *strings* into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

xstr –c name

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file **x.c**, to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file xs.c declaring the common *xstr* space can be created by a command of the form

xstr -c name1 name2 name3 ...

This **xs.c** file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

xstr can also be used on a single file. A command

xstr name

creates files **x.c** and **xs.c** as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *xstr* reads from its standard input when the argument – is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking xs.o unless truly necessary.

Files

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array "xstr"
/tmp/xs*	Temp file when "xstr name" doesn't touch *strings*

See Also

mkstr(CP)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* , both strings will be placed in the data base when just placing the longer one there will do.

Name

yacc – Invokes a compiler-compiler.

Syntax

yacc [–vd] grammar

Description

yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex* (CP) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

Files

y.output	
y.tab.c	
y.tab.h	Defines for token names
yacc.tmp, yacc.acts	Temporary files
/usr/lib/yaccpar	Parser prototype for C programs

See Also

lex(CP)

Diagnostics

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

Notes

Because filenames are fixed, at most one *yacc* process can be active in a given directory at a time.

Contents

System Service (S)

getpid, getpgrp, getppid	Gets process, process group, and parent process IDs.
getpw	Gets password for a given user ID.
getpwent, getpwuid, getpwnam, setpwent, endpwent	Gets password file entry.
gets, fgets	Gets a string from a stream.
getuid, geteuid, getgid, getegid	Gets real user, effective user, real group, and effective group IDs.
getut	Accesses utmp file entry.
hsearch	Manages hash search tables.
hypot, cabs	Determines Euclidean distance.
ioctl	Controls character devices.
kill	Sends a signal to a process or a group of processes.
l3tol, ltol3	Converts between 3-byte integers and long integers.
link	Links a new filename to an existing file.
lock	Locks a process in primary memory.
lockf	Provide semaphores and record locking in files.
locking	Locks or unlocks a file region for reading or writing.
logname	Finds login name of user.
lsearch	Performs linear search and update.
lseek	Moves read/write file pointer.
malloc, free, realloc, calloc	Allocates main memory.
matherr	Error handling function.
memory	Memory operations.
mknod	Makes a directory, or a special or ordinary file.
mktemp	Makes a unique filename.
monitor	Prepares execution profile.
mount	Mounts a file system.
msgctl	Message control operations.
msgget	Message queue.
msgop	Message operations.
nap	Suspends execution for a short interval.
nice	Changes priority of a process.
nlist	Gets entries from name list.
open	Opens file for reading or writing.
opensem	Opens a semaphore.
pause	Suspends a process until a signal occurs.
perror, sys_errlist, sys_nerr, errno	Sends system error messages.

pipe	Creates an interprocess pipe.
plock	Lock process, text, or data in memory.
popen, pclose	Initiates I/O to or from a process.
printf, fprintf, sprintf	Formats output.
proctl	Controls processes or process groups.
profil	Creates an execution time profile.
ptrace	Traces a process.
putc, putchar, fputc, putw	Puts a character or word on a stream.
putenv	Changes or adds environment variable.
putpwent	Writes a password file entry.
puts, fputs	Puts a string on a stream.
qsort	Performs a sort.
rand, srand	Generates a random number.
rdchk	Checks to see if there is data to be read.
read	Reads from a file.
regex, regcmp	Compiles and executes regular expressions.
regexp	Regular expression compile and match routines.
sbrk, brk	Changes data segment space allocation.
scanf, fscanf, sscanf	Converts and formats input.
sdenter, sdleave	Synchronizes access to a shared data segment.
sdget	Attachs and detachs a shared data segment.
sdgetv, sdwaitv	Synchronizes shared data access.
semctl	Semaphore control.
semget	Semaphores, gets set.
semop	Semaphore operations.
setbuf	Assigns buffering to a stream.
setjmp, longjmp	Performs a nonlocal "goto".
setpgrp	Sets process group ID.
setuid, setgid	Sets user and group IDs.
shmctl	Shared memory control.
shmget	Shared memory, gets.
shmop	Shared memory operations.
shutdn	Flushes block I/O and halts the CPU.
signal	Specifies what to do upon receipt of a signal.
sigsem	Signals a process waiting on a semaphore.
sinh, cosh, tanh	Performs hyperbolic functions.
sleep	Suspends execution for an interval.
sputl	Accesses long integer data.
ssignal, gsignal	Implements software signals.
stat, fstat	Gets file status.
stdio	Performs standard buffered input and output.
stdipc	Standard interprocess communications package.
stime	Sets the time.

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strdup	Perform string operations.
strtod	Converts string to double precision numbers.
strtol	String to integer.
swab	Swaps bytes.
swapadd	Adds swap area.
sync	Updates the super-block.
system	Executes a shell command.
termcap, tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs	Performs terminal functions.
terminfo	Terminal description database.
time, ftime	Gets time and date.
times	Gets process and child process times.
tmpfile	Creates a temporary file.
tmpnam	Creates a name for a temporary file.
trig, sin, cos, tan, asin, acos, atan, atan2	Performs trigonometric functions.
tsearch	Manages binarys search trees.
ttyname, isatty	Finds the name of a terminal.
ttyslot	Finds the slot in the utmp file of the current user.
uadmin	Administrative control.
ulimit	Gets and sets user limits.
umask	Sets and gets file creation mask.
umount	Unmounts a file system.
uname	Gets name of current XENIX system.
ungetc	Pushes character back into input stream.
unlink	Removes directory entry.
ustat	Gets file system statistics.
utime	Sets file access and modification times.
varargs	Variable argument list.
vprintf	Prints formatted output of a varargs argument list.
wait	Waits for a child process to stop or terminate.
waitsem, nbwaitsem	Awaits and checks access to a resource governed by a semaphore.
write	Writes to a file.
xlist, fxlist	Gets name list entries from files.

Name

intro – Introduces system services, library routines and error numbers.

Syntax

#include <errno.h>

Description

This section describes all system services. System services include all routines or system calls that are available in the operating system kernel. These routines are available to a C program automatically as part of the standard library **libc**. Other routines are available in a variety of libraries. On 8086/88, and 286 systems, versions for Small, Middle, and Large model programs are provided (that is, three of each library). On 386 systems, Small, Middle, and Large programs for 286 processes and Small model programs for 386 processes are provided.

To use routines in a program that are not part of the standard library **libc**, the appropriate library must be linked. This is done by specifying **-l name** to the compiler or linker, where *name* is the name listed below. For example **-l m** , and **-l termcap** are specifications to the linker to search the named libraries for routines to be linked to the object module. The names of the available libraries are:

c The standard library containing all system call interfaces, Standard I/O routines, and other general purpose services.

m The standard math library.

termcap
Routines for accessing the *termcap* data base describing terminal characteristics.

curses Screen and cursor manipulation routines.

dbm Data base management routines.

x The standard XENIX library.

Most services that are part of the operating system kernel have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always −1; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 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.

2 ENOENT No such file or directory:
This error occurs when a filename is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process:
No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

4 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.

5 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.

6 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 on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long:
An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

8 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*(F)).

9 EBADF Bad file number:
Either a file descriptor refers to no open file, or a read (respectively write) request is made to a file which is open only for writing (respectively reading).

10 ECHILD No child processes:
A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 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.

12 ENOMEM Not enough space:
During an *exec*, or *sbrk*, a program asks for more space than the system is able to supply. This is not 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*.

13 EACCES Permission denied:
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address:
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required:
A nonblock file was mentioned where a block device was required, e.g., in *mount*.

16 EBUSY Device 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.

17 EEXIST File exists:
An existing file was mentioned in an inappropriate context, e.g., *link*.

18 EXDEV Cross-device link:
A link to a file on another device was attempted.

19 ENODEV No such device:
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory:
A nondirectory was specified where a directory is required, for example, in a path prefix or as an argument to *chdir*(S).

21 EISDIR Is a directory:
An attempt to write on a directory.

22 EINVAL Invalid argument:
An invalid argument (e.g., dismounting a nonmounted 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 (S) entries of this manual.

23 ENFILE File table overflow:
The system's table of open files is full and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files:
No process may have more than 60 file descriptors open at a time.

25 ENOTTY Not a character device

26 ETXTBSY Text file busy:
An attempt to execute a pure-procedure program which is currently open for writing (or reading). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large:
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(S).

28 ENOSPC No space left on device:
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek:
An *lseek* was issued to a pipe.

30 EROFS Read-only file system:
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links:
An attempt to make more than the maximum number of links (1000) to a file.

32 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.

33 EDOM Math arg out of domain of func:
The argument of a function in the math package is out of the domain of the function.

34 ERANGE Math result not representable:
The value of a function in the math package is not representable within machine precision.

35 EUCLEAN File system needs cleaning:
An attempt was made to *mount*(S) a file system whose super-block is not flagged clean.

36 EDEADLOCK Would deadlock:
A process' attempt to lock a file region would cause a deadlock between processes vying for control of that region.

36 EDEADLK Would deadlock:
A process' attempt to lock a file region would cause a deadlock between processes vying for control of that region.

37 ENOTNAM Not a name file:
A *creatsem*(S), *opensem*(S), *waitsem*(S), or *sigsem*(S) was issued using an invalid semaphore identifier.

38 ENAVAIL Not available:
An *opensem*(S), *waitsem*(S) or *sigsem*(S) was issued to a semaphore that has not been initialized by a call to *creatsem*(S). A *sigsem* was issued to a semaphore out of sequence; i.e., before the process has issued the corresponding *waitsem* to the semaphore. An *nbwaitsem* was issued to a semaphore guarding a resource that is currently in use by another process. The semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exits without relinquishing control properly; i.e., without issuing a *waitsem* on the semaphore.

39 EISNAM A name file:
A name file (semaphore, shared data, etc.) was specified when not expected.

43 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*(S).

44 EIDRM Identifier removed:
This error is returned to a process that resumes execution due to the removal of an identifier from the file system's name space; see *msgctl*(S), *semctl*(S), and *shmctl*(S).

45 ENOLCK No locks available:
The system's lock table was full, and a file locking or unlocking operation was attempted which would have created an additional lock table entry.

Definitions

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID

A new process is created by a currently active process; see *fork*(S). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(S).

Process Group Leader

A process group leader is any process whose process group ID is the same as its process ID . Any process may become a group leader by calling *setgrp*(S). A process inherits the process group ID of the process that created it, see *fork*(S) and *exec*(S).

TTY Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the TTY group ID. This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(S) and *signal*(S).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and a real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process' real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec* (S).

Super-User

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

proc0 is the scheduler. *proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

Filename

Names consisting of up to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for a slash (/).

Note that it is generally unwise to use *, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell. Likewise, the high order bit of the character should not be set.

Pathname and Path Prefix

A pathname is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a filename. A filename is a string of 1 to 14 characters other than the ASCII slash and null, and a directory name is a string of 1 to 14 characters (other than the ASCII slash and null) naming a directory.

If a pathname begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null pathname is treated as if it named a nonexistent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as "dot" and "dot-dot" respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

Each process has a concept of a root directory and a current working directory for the purpose of resolving pathname searches associated with it. A process' root directory need not be the root directory of the root file system. See *chroot*(C) and *chroot*(S).

File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

> The process' effective user ID is super-user.
>
> The process' effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.
>
> The process' effective user ID does not match the user ID of the owner of the file, and the process' group ID matches the group of the file, and the appropriate access bit of the "group" portion (070) of the file mode is set.
>
> The process' effective user ID does not match the user ID of the owner of the file, and the process' effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied. See *chmod*(C) and *chmod*(S).

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(S) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;  /* operation permission struct */
ushort   msg_qnum;           /* number of msgs on q */
ushort   msg_qbytes;         /* max number of bytes on q */
ushort   msg_lspid;          /* pid of last msgsnd operation */
ushort   msg_lrpid;          /* pid of last msgrcv operation */
time_t   msg_stime;          /* last msgsnd time */
time_t   msg_rtime;          /* last msgrcv time */
time_t   msg_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). The structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/w permission */
```

msg_qnum is the number of messages currently on the queue. **msg_qbytes** is the maximum number of bytes allowed on the queue. **msg_lspid** is the process ID of the last process that performed a *msgsnd* operation. **msg_lrpid** is the process ID of the last process that performed a *msgrcv* operation. **msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(S) operation that changed a member in the above structure.

Message Operation Permissions

In the *msgop*(S) and *msgctl*(S) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed. It is interpreted as follows:

00400	Read by user

00200	Write by user
00060	Read, write by group
00006	Read, write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*, and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.uid** or **msg_perm.cuid** and the effective group ID of the process matches **msg_perm.gid** or **msg_perm.cgid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.uid** or **msg_perm.cuid** and the effective group ID of the process does not match **msg_perm.gid** or **msg_perm.cgid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(S) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;  /* operation permission struct */
ushort   sem_nsems;          /* number of sems in set */
time_t   sem_otime;          /* last operation time */
time_t   sem_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   cuid;       /* creator user id */
ushort   cgid;       /* creator group id */
ushort   uid;        /* user id */
ushort   gid;        /* group id */
ushort   mode;       /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a "sem_num" . Sem_num values run sequentially from 0 to the value of **sem_nsems** minus 1. **sem_otime** is the time of the last *semop*(S) operation, and **sem_ctime** is the time of the last *semctl*(S) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort  semval;     /* semaphore value */
short   sempid;     /* pid of last operation  */
ushort  semncnt;    /* # awaiting semval > cval */
ushort  semzcnt;    /* # awaiting semval = 0 */
```

semval is a non-negative integer. **sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. **semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions

In the *semop*(S) and *semctl*(S) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed and is interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, alter by group
00006	Read, alter by others

Read and alter permissions for a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid,* and the appropriate "user" portion (0600) bit of **sem_perm.mode** is set.

The effective user ID of the process does not match **sem_perm.uid** , or **sem_perm.cuid** and the effective group ID of the process matches **sem_perm.gid** or **sem_perm.cgid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

> The effective user ID of the process does not match **sem_perm.uid** or **sem_perm.cuid** and the effective group ID of the process does not match **sem_perm.gid** or **sem_perm.cgid** and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(S) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct   ipc_perm shm_perm;  /* operation permission struct */
int      shm_segsz;          /* size of segment */
ushort   shm_cpid;           /* creator pid */
ushort   shm_lpid;           /* pid of last operation */
short    shm_nattch;         /* number of current attaches */
time_t   shm_atime;          /* last attach time */
time_t   shm_dtime;          /* last detach time */
time_t   shm_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). The structure includes the following members:

```
ushort   cuid;        /* creator user id */
ushort   cgid;        /* creator group id */
ushort   uid;         /* user id */
ushort   gid;         /* group id */
ushort   mode;        /* r/w permission */
```

shm_segsz specifies the size of the shared memory segment. **shm_cpid** is the process ID of the process that created the shared memory identifier. **shm_lpid** is the process ID of the last process that performed a *shmop*(S) operation. **shm_nattch** is the number of processes that currently have this segment attached. **shm_atime** is the time of the last *shmat* operation. **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(S) operation that changed one of the above structure members.

Shared Memory Operation Permissions

In the *shmop*(S) and *shmctl*(S) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed. It is interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, write by group
00006	Read, write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.uid** or **shm_perm.cuid** and the effective group ID of the process matches **shm_perm.gid** or **shm_perm.cgid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.uid** or **shm_perm.cuid** and the effective group ID of the process does not match **shm_perm.gid** or **shm_perm.cgid** and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

See Also

close(S), ioctl(S), open(S), pipe(S), read(S), write(S)

Name

a64l, l64a – Converts between long integer and base 64 ASCII.

Syntax

long a64l (s)
char *s;

char *l64a (l)
long l;

Description

These routines are used to maintain numbers stored in base 64 ASCII. 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 through 11, **A** through **Z** for 12 through 37, and **a** through **z** for 38 through 63.

a64l takes a pointer to a null-terminated base 64 representation and returns a corresponding **long** value. *l64a* takes a **long** argument and returns a pointer to the corresponding base 64 representation.

Notes

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

Name

abort – Generates an IOT fault.

Syntax

int abort ( )

Description

abort first closes all open files, if possible, then causes an I/O trap signal (SIGIOT) to be sent to the calling process. This usually results in termination with a core dump.

abort can return control if the calling process is set to catch or ignore the SIGIOT signal; see *signal*(S).

See Also

adb(CP), exit(S), signal(S)

Diagnostics

If an aborted process returns control to the shell (*sh* (C)), the shell usually displays the message "abort – core dumped".

Name

abs – Returns an integer absolute value.

Syntax

int abs (i)
int i;

Description

abs returns the absolute value of its integer operand.

See Also

fabs in floor(S)

Notes

If the largest negative integer supported by the hardware is given, the function returns it unchanged.

Name

access – Determines accessibility of a file.

Syntax

int access (path, amode)
char *path;
int amode;

Description

path points to a pathname 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 for *amode* can be formed by adding any combination of the following:

04	Read
02	Write
01	Execute (search)
00	Check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null pathname. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

path points outside the process' allocated address space. [EFAULT]

access checks the permissions for the owner of a file by checking the "owner" read, write, and execute mode bits. For members of the file's group, the "group" mode bits are checked. For all others, the "other" mode bits are checked.

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.

See Also

chmod(S), stat(S)

Notes

The super-user (root) may access any file, regardless of permission settings.

Name

acct – Enables or disables process accounting.

Syntax

#include <sys/types.h>

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. A process can be terminated by a call to *exit* or by receipt of a signal which it does not ignore or catch; see *exit*(S) and *signal*(S). The effective user ID of the calling process must be super-user to use this call.

path points to the pathname of the accounting file. The accounting file format is given in *acct*(F).

The accounting routine is enabled if *path* is nonzero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

acct will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's pathname do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EACCES]

The named file resides on a read-only file system. [EROFS]

path points to an illegal address. [EFAULT]

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

accton(C), acctcom(C), acct(F)

Name

alarm – Sets a process' alarm clock.

Syntax

unsigned alarm (sec)
unsigned sec;

Description

alarm sets the calling process' alarm clock to *sec* seconds. After *sec* "real-time" seconds have elasped, the alarm clock sends a SIGALRM signal to the process; see *signal*(S).

Although *alarm* does not wait for the signal after setting the alarm clock, *pause*(S) may be used to make the calling process wait.

Alarm requests are not stacked; successive calls reset the calling process' alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

fork(S) sets the alarm clock of a new process to 0. a process created by *exec*(S) inherits the time left on the old process's alarm clock.

Return Value

alarm returns the amount of time previously remaining in the calling process' alarm clock.

See Also

pause(S), signal(S)

Name

assert – Helps verify validity of program.

Syntax

#include <stdio.h>
#include <assert.h>

void assert (expression)
int expression;

Description

This macro is useful for putting diagnostics into programs under development. When it is executed, if *expression* is false (zero), it displays:

Assertion failed: *expression*, file *name*, line *nnn*

on the standard error file and aborts. *name* is the source filename and *nnn* is the source line number of the *assert* statement.

Notes

To suppress calls to *assert*, use the **-DNDEBUG** option (see *cpp*(CP)), or insert the preprocessor control statement, **#define NDEBUG** before the **#include <assert.h>** statement when compiling the program.

See Also

abort(S), cpp(CP)

Name

atof, atoi, atol – Converts ASCII to numbers.

Syntax

double atof (nptr)
char *nptr;

int atoi (nptr)
char *nptr;

long atol (nptr)
char *nptr;

Description

These functions convert a string pointed to by *nptr* to floating, integer, and long integer numbers respectively. The first unrecognized character ends the string.

atof recognizes a string of the form:

[+ | –] digits[. digits][e| E [+ | –] digits]

where the digits are continguous decimal digits. Any number of tabs and spaces may precede the string. The **+** and **–** signs are optional. Either **e** or **E** may be used to mark the beginning of the exponent.

atoi and *atol* recognize strings of the form:

[+ | –] digits

where the digits are contiguous decimal digits. Any number of tabs and spaces may precede the string. The **+** and **–** signs are optional.

See Also

scanf(S)

Notes

There are no provisions for overflow.

These routines must be linked by using the **–lm** linker option.

Name

bessel, j0, j1, jn, y0, y1, yn – Performs Bessel functions.

Syntax

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
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. The value of x must be positive.

y0 and *y1* return 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.

See Also

matherr(S)

Diagnostics

Negative arguments cause *y0*, *y1*, and *yn* to return a -**HUGE** value and to set *errno* to **EDOM**. In addition, a message indicating DOMAIN error is displayed on the standard error output. Arguments too large in magnitude cause *j0*, *j1*, and *y1* to return zero and to set *errno* to **ERANGE**. In addition, a message indicating

TLOSS error is displayed on the standard error output. These error-handling procedures can be changed with the *matherr*(S) function.

Notes

These routines must be linked by using the **-lm** linker option.

Name

brkctl – Allocates data in a far segment.

Syntax

#include <sys/brk.h>

char far *brkctl(*command*, *increment*, *ptr*)
int *command*;
long *increment*;
char far *ptr;

Description

The *brkctl* system call allocates and deallocates memory in additional data segments in small and middle model programs. In order for the C compiler to make use of the return values in small and middle model programs, *brkctl* must be declared to return a far pointer. To enable the 'far' keyword for small model C programs, the **–Me** option to the compiler must be used. Middle model C programs require the **–Mme** option.

command is either **BR_ARGSEG**, **BR_NEWSEG**, or **BR_IMPSEG**.

increment is a signed long increment. If positive, it must be less than 64K; if negative, its absolute value must be less than the sum of the total memory in all far segments plus the amount allocated in the near segment after process creation.

ptr is used only when *command* is **BR_ARGSEG**.

If *increment* is positive, *brkctl* returns a far pointer to the base of at least *increment* number of bytes of memory (see box on next page).

If the *command* is **BR_IMPSEG**, and a negative *increment* causes one or more segments to be freed, the 'segment in question' (see the *Return Values* section) is the last remaining segment that was not freed. **BR_IMPSEG** implies the use of the last data segment. Unless the process is small or middle model and currently has only one data segment, a positive *increment* that would overflow the last data segment causes a new segment to be allocated.

If the *command* is **BR_ARGSEG**, the *increment* may not be more negative than the size of the segment. The third argument (*ptr*), is assumed to be a far pointer in all models; the offset portion is never used.

If the *command* is **BR_NEWSEG**, the *increment* may not be negative at all. Any memory allocated is guaranteed to be at the base of a new segment.

Return Value

brkctl() almost always returns a far pointer to the base of the affected region, (char far *)-1 on error.

When the *increment* is greater than 0, the return value is a pointer to the base of the newly allocated memory.

When the *increment* is less than or equal to 0, the return value is a pointer to the first illegal byte in the segment in question (usually the base of the deallocated memory). If that segment is full (exactly 64K bytes), the return value will be a pointer to the base of the next segment (which may or may not exist).

Command	Increment	Ptr	Action
BR_ARGSEG	0	<valid far ptr>	report on segment
BR_ARGSEG	other	<valid far ptr>	increment specified segment
BR_NEWSEG	0	-	allocate new segment, size = 0
BR_NEWSEG	other	-	allocate new segment, size = increment
BR_IMPSEG	0	-	report on last segment; may free up empty segment(s).
BR_IMPSEG	other	-	increment last segment; on large model (or small and middle model with mutiple data segments) may allocate new segment.

See Also

cc(CP), ld(CP), machine(M), malloc(S), sbrk(S)

Notes

The *brkctl* system call should be used only for dynamically allocating additional segments in small and middle model programs. All other uses should be avoided in favor of *sbrk*(S), *malloc*(S), and other standard UNIX system services. The functionality of *brkctl* may change in future releases.

brkctl is currently available only on protected mode XENIX.

In all models, the 'near' data segment must be the first data segment.

brkctl calls with **BR_IMPSEG** and a negative *increment* that would affect a shared data segment are refused.

Name

bsearch – Performs a binary search.

Syntax

```
#include <search.h>

char *bsearch (key, base, nel, width, compar)
char *key;
char *base;
unsigned nel, width;
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 the location at which a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function, *compar*. *key* is a pointer to the datum to be located in the table. *base* is a pointer to the elements at the base of the table. *nel* is the number of elements in the table. *width* is the size of an element in bytes. *compar* is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than zero, depending on whether 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. The nodes consist of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

The following code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message, (as shown on the next page).

```
#include <stdio.h>
#include <search.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);
}
```

See Also

hsearch(S), lsearch(S), qsort(S), tsearch(S)

Diagnostics

If the key cannot be found in the table, a NULL (0) 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 pointer-to-element.

Name

chdir – Changes the working directory.

Syntax

int chdir (path)
char *path;

Description

path points to the pathname of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for pathnames not beginning with **/**.

chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the pathname. [EACCES]

path points outside the process' allocated address space. [EFAULT]

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

chroot(S)

Name

chmod – Changes mode of a file.

Syntax

int chmod (path, mode)
char *path;
int mode;

Description

path points to a pathname naming a file. *chmod* sets the access permission portion of the named file's mode. It sets the access permission portion according to the bit pattern contained in *mode*.

Access permission bits for *mode* can be formed by adding any combination of the following:

04000 Set user ID on execution
02000 Set group ID on execution
01000 Save text image after execution
00400 Read by owner
00200 Write by owner
00100 Execute (or search if a directory) by owner
00040 Read by group
00020 Write by group
00010 Execute (or search) by group
00004 Read by others
00002 Write by others
00001 Execute (or search) by others

To change the mode of a file, the effective user ID of the process must match the owner of the file or must be super-user.

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 or the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing, when its last user terminates, mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file. Thus, when the next user executes the file, the text need not be read from the file system but can simply be swapped in, saving time. Many systems have relatively small amounts of swap space, and the same-text bit should be used sparingly, if at all.

chmod will fail and the file mode will be unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

path points outside the process' allocated address space. [EFAULT]

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

chown(S), mknod(S)

Name

chown – Changes the owner and group of a file.

Syntax

int chown (path, owner, group)
char *path;
int owner, group;

Description

path points to a pathname naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with an effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file, and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

path points outside the process' allocated address space. [EFAULT]

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

chmod(S)

Name

chroot – Changes the root directory.

Syntax

int chroot (path)
char *path;

Description

path points to a pathname naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for pathnames beginning with **/**. The user's working directory is unaffected by the *chroot* system call.

To change the root directory, the effective user ID of the process must be super-user.

The ".." entry in the root directory is interpreted to mean the root directory itself. Thus, ".." cannot be used to access files outside the root directory.

chroot will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [EPERM]

path points outside the process' allocated address space. [EFAULT]

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

chdir(S), chroot(C)

Name

chsize – Changes the size of a file.

Syntax

int chsize (fildes, size)
int fildes;
long size;

Description

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *chsize* changes the size of the file associated with the file descriptor *fildes* to be exactly *size* bytes in length. The routine either truncates the file, or pads it with an appropriate number of bytes. If *size* is less than the initial size of the file, then all allocated disk blocks between *size* and the initial file size are freed.

The maximum file size as set by *ulimit* (S) is enforced when *chsize* is called, rather than on subsequent writes. Thus *chsize* fails, and the file size remains unchanged if the new changed file size would exceed the *ulimit*.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, the value -1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), lseek(S), open(S), pipe(S), ulimit(S)

Notes

In general if *chsize* is used to expand the size of a file, when data is written to the end of the file, intervening blocks are filled with zeros. In a few rare cases, reducing the file size may not remove the data beyond the new end-of-file. This routine must be linked with the linker option **-lx**.

Name

clock – Reports CPU time used.

Syntax

long clock ( )

Description

clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The reported time equals the sum of user and system times of the calling process and any terminated child processes for which *wait* or *system*(S) were executed.

The resolution of the clock is machine dependent. Refer to the manual page *machine*(HW) for the clock resolution on your system.

See Also

machine(HW), system(S), times(S), wait(S)

Notes

The microsecond value returned by *clock* is compatible 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).

Name

close – Closes a file descriptor.

Syntax

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 outstanding record locks on the file indicated by *fildes* that are owned by the calling process are removed.

close will fail if *fildes* is not a valid open file descriptor. [EBADF]

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(S), dup(S), exec(S), fcntl(S), open(S), pipe(S)

Name

conv, toupper, tolower, toascii – Translates characters.

Syntax

```
#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* convert the argument *c* to a letter of opposite case. Arguments may be the integers −1 through 255 (the same values returned by *getc*(S)). If the argument of *toupper* represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of *tolower* represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted argument values and are faster. *_toupper* requires a lowercase letter as its argument; its result is the corresponding uppercase letter. *_tolower* requires an uppercase letter as its argument; its result is the corresponding lowercase letter. All other arguments cause unpredictable results.

toascii converts integer values to ASCII characters. The function clears all bits of the integer that are not part of a standard ASCII character; it is intended for compatibility with other systems.

See Also

ctype(S)

Notes

Because *_toupper* and *_tolower* are implemented as macros, they should not be used where unwanted side effects may occur. Removing the *_toupper* and *_tolower* macros with the **#undef** directive causes the corresponding library functions to be linked instead. This allows any arguments to be used without worry about side effects.

Name

creat – Creates a new file or rewrites an existing one.

Syntax

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 pathname 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 process' effective user ID, the file's group ID is set to the process' effective group ID, and the access permission bits (i.e., the low-order 12 bits of the file mode) are set to the value of *mode*. *mode* may have the same values as described for *chmod*(S). *creat* will then modify the access permission bits as follows:

All bits set in the process' file mode creation mask are cleared. See *umask*(S).

The "save text image after execution bit" is cleared. See *chmod*(S).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, 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*(S). No process may have more than 60 files open simultaneously. A new file may be created with a *mode* that forbids writing.

creat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The pathname is null. [ENOENT]

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

The named file resides or would reside on a read-only file system. [EROFS]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The file exists and write permission is denied. [EACCES]

The named file is an existing directory. [EISDIR]

Sixty file descriptors are currently open. [EMFILE]

path points outside the process' allocated address space. [ENOSPC]

The directory to contain the file cannot be extended. [EFAULT]

The system file table is full. [ENFILE]

Return Value

Upon successful completion, a nonnegative 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(S), dup(S), lseek(S), open(S), read(S), umask(S), write(S)

Notes

open (S) is preferred to *creat*.

Name

creatsem – Creates an instance of a binary semaphore.

Syntax

int = creatsem(sem_name,mode)
char *sem_name;
int mode;

Description

creatsem defines a binary semaphore named by *sem_name* to be used by *waitsem*(S) and *sigsem*(S) to manage mutually exclusive access to a resource, shared variable, or critical section of a program. *creatsem* returns a unique semaphore number, *sem_num*, which may then be used as the parameter in *waitsem* and *sigsem* calls. Semaphores are special files of 0 length. The filename space is used to provide unique identifiers for semaphores. *mode* sets the accessibility of the semaphore using the same format as file access bits. Access to a semaphore is granted only on the basis of the read access bit; the write and execute bits are ignored.

A semaphore can be operated on only by a synchronizing primitive, such as *waitsem* or *sigsem*, by *creatsem* which initializes it to some value, or by *opensem* which opens the semaphore for use by a process. Synchronizing primitives are guaranteed to be executed without interruption once started. These primitives are used by associating a semaphore with each resource (including critical code sections) to be protected.

The process controlling the semaphore should issue:

```
sem_num = creatsem("semaphore", mode);
```

to create, initialize, and open the semaphore for that process. All other processes using the semaphore should issue:

```
sem_num = opensem("semaphore");
```

to access the semaphore's identification value. Note that a process cannot open and use a semaphore that has not been initialized by a call to *creatsem*, nor should a process open a semaphore more than once in one period of execution. Both the creating and opening processes use *waitsem* and *sigsem* to use the semaphore *sem_num*.

Compatibility

creatsem can only be used to define XENIX version 3.0 semaphores, not XENIX System V semaphores.

See Also

opensem(S), waitsem(S), sigsem(S)

Diagnostics

creatsem returns the value −1 if an error occurs. If the semaphore named by *sem_name* is already open for use by other processes, *errno* is set to EEXIST. If the file specified exists but is not a semaphore type, *errno* is set to ENOTNAM. If the semaphore has not been initialized by a call to *creatsem*, *errno* is set to ENAVAIL.

Notes

After a *creatsem* you must do a *waitsem* to gain control of a given resource.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This function must be linked with the linker option **-lx**.

Name

ctermid – Generates a filename for a terminal.

Syntax

#include <stdio.h>

char *ctermid(s)
char *s;

Description

ctermid returns a pointer to a string that, when used as a filename, refers to the controlling terminal of the calling process.

If (int)*s* is zero, 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. If (int)*s* is nonzero, then *s* is assumed to point to a character array of at least **L_ctermid** elements; the string is placed in this array and the value of *s* is returned. The manifest constant **L_ctermid** is defined in **<stdio.h>**.

Notes

The difference between *ctermid* and *ttyname*(S) is that *ttyname* must be given a file descriptor and it returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a magic string (**/dev/tty**) that will refer to the terminal if used as a filename. Thus *ttyname* is useless unless the process already has at least one file open to a terminal.

See Also

ttyname(S)

Name

ctime, localtime, gmtime, asctime, tzset – Converts date and time to ASCII.

Syntax

```
char *ctime (clock)
long *clock;

#include <time.h>
#include <sys/types.h>

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

void tzset ( )

extern long timezone;
extern long altzone;
extern int daylight;
extern char *tzname[2];
```

Description

ctime converts a time pointed to by *clock* (such as returned by *time*(S)) into ASCII and returns a pointer to a 26-character string in the following form:

Sun Sep 16 01:03:52 1973\n\0

If necessary, fields in this string are padded with spaces to keep the string a constant length.

localtime and *gmtime* return pointers to structures containing the time as a variety of individual quantities. These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year (since 1900), day of year (0-365), seconds from GMT (East < 0), a flag that is nonzero if summer time (daylight saving time) is in effect, and the name of the timezone. *localtime* corrects for the time zone and possible summer time. *gmtime* converts directly to Greenwich time (GMT), which is the time the XENIX system uses.

asctime converts the times returned by *localtime* and *gmtime* to a 26-character ASCII string and returns a pointer to this string.

The structure declaration for *tm* is defined in **/usr/include/time.h.**

The external long variable *timezone* contains the difference, in seconds, between GMT and local standard time (e.g., in Eastern Standard Time (EST), *timezone* is 5*60*60); similarly, the external long variable *altzone* contains the difference, in seconds, between GMT and local summer time (e.g., in Eastern Daylight Time (EDT), *altzone* is 4*60*60); the external integer variable *daylight* is nonzero if and only if summer time conversion should be applied.

If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the default time zone as determined by *ftime*() (see *time*(S)). The value of TZ is described in detail on the *tz*(M) manual page. The effects of setting TZ are thus to change the values of the external variables *timezone*, *altzone*, and *daylight*. In addition, the time zone names contained in the external variable

```
char *tzname[2] = {"EST", "EDT"};
```

are set from the environment variable. The rule for when to change between standard time and summer time can be specified in the TZ string. If a rule is not specified, the standard U.S.A. Daylight Savings Time conversion is applied. The program knows about the peculiarities of this conversion in 1974 and 1975 and the change in 1987. The function *tzset* sets the external variables from TZ ; it is called by *asct* and may also be called explicitly by the user.

See Also

environ(M), getenv(S), time(S), tz(M)

Notes

The return values point to static data, whose content is overwritten by each call.

Changes to *TZ* are immediately effective, (i.e. if a process changes the *TZ* variable, the next call to a *ctime*(S) routine returns a value based on the new value of the variable).

Name

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii, tolower, toupper, toascii – Classifies or converts characters.

Syntax

#include <ctype.h>

int isalpha (c)
int c;

. . .

Description

These macros classify ASCII-coded integer values by table lookup. Each returns 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 (see *stdio* (S)).

isalpha	*c* is a letter
isupper	*c* is an uppercase letter
islower	*c* is a lowercase letter
isdigit	*c* is a digit [0-9]
isxdigit	*c* is a hexidecimal digit [0-9], [A-F] or [a-f]
isalnum	*c* is an alphanumeric
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or form feed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, octal 40 (space) through octal 176 (tilde)
isgraph	*c* is a printing character, like *isprint* except false for space

iscntrl c is a delete character (octal 177) or ordinary control character (less than octal 40).

isascii c is an ASCII character, code less than 0200

If the argument to any of these macros is not in the domain of the function, the result is undefined.

The following macros convert to ASCII-coded integer values. *tolower* and *toupper* are implemented as macros, but can be undefined to get non-macro versions from **libc**. Non-alphabetic values passed to *toupper* and *tolower* will be returned unchanged.

tolower
If *c* is an uppercase letter, it is returned as a lowercase letter

toupper
If *c* is a lowercase letter, it is returned as an uppercase letter

toascii
c is truncated to the lowest 7 bits

See Also

ascii(M)

Name

curses – Performs screen and cursor functions.

Syntax

#include <curses.h>
WINDOW *curscr, *stdscr;

cc −DM_TERMCAP *filename* **−ltcap −ltermlib**

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, **curscr**. The user modifies this image by modifying the standard screen, **stdscr**, or by setting up a new screen. The *refresh* and *wrefresh* routines make the current screen look like the modified one. 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 routines are linked with the linker options **−ltcap** and **−ltermlib**. Programs using these routines must be compiled with **M_TERMCAP** defined.

Functions

int addch(ch)
char ch;

Adds a character to **stdscr**

int addstr(str)
char *str;

Adds a string to **stdscr**

int box(win,vert,hor)
WINDOW *win;
char vert, hor;

Draws a box around a window

int crmode()

Sets cbreak mode

int clear()

Clears **stdscr**

int clearok(win,state)
WINDOW *win;
bool state;
Sets clear flag for *win*

int clrtobot()
Clears to bottom on **stdscr**

int clrtoeol()
Clears to end of line on **stdscr**

int delch()
Deletes character from **stdscr**

int deleteln()
Deletes line from **stdscr**

int delwin(win)
WINDOW *win;
Delete *win*

int echo()
Sets echo mode

int endwin()
Terminates screen processing

int erase()
Erase **stdscr**

int getch()
Gets a char through **stdscr**

int getstr(str)
char *str;
Gets a string through **stdscr**

int gettmode()
Gets tty modes

int getyx(win,y,x)
WINDOW *win;
int y,x;
Gets current (y,x) position of *win*

int inch()
Gets char at current (y,x) co-ordinates

WINDOW *initscr()
Initializes screens

int insch(c)
char c;
Inserts character in **stdscr**

int insertln()
Inserts blank line in **stdscr**

int leaveok(win,state)
WINDOW *win;
bool state;
Sets leave flag for *win*

int longname(termbuf,name)
char *termbuf, *name;
Gets long name from *termbuf*

int move(y,x)
int y,x;
Moves to (y,x) on **stdscr**

int mvaddch(y,x,ch)
int y,x;
char ch;
Moves to (y,x) and adds character *ch*

int mvaddstr(y,x,str)
int y,x;
char *str;
Moves to (y,x) and adds string *str*

int mvcur(lasty,lastx,newy,newx)
int lasty, lastx, newy, newx;
Moves cursor the from (lasty,lastx) to (newy,newx)

int mvdelch(y,x)
int y,x;
Moves to (y,x) and deletes character from **stdscr**

int mvgetch(y,x)
int y,x;
Moves to (y,x) and gets a char through **stdscr**

int mvgetstr(y,x,str)
int y,x;
char *str;
Moves to (y,x) and gets a string through **stdscr**

int mvinch(y,x)
int y,x;
Moves to (y,x) and gets char at current co-ordinates

int mvinsch(y,x,c)
int y,x;
char c;
Moves to (y,x) and inserts character in stdscr

int mvwaddch(win, y,x,ch)
WINDOW *win;
int y,x;
char ch;
Moves to (y,x) in *win* and adds character *ch*

int mvwaddstr(win,y,x,str)
WINDOW *win;
int y,x;
char *str;
Moves to (y,x) in *win* and adds string *str*

int mvwdelch(win,y,x)
WINDOW *win;
int y,x;
Moves to (y,x) in *win* and deletes the character

int mvwgetch(win,y,x)
WINDOW *win;
int y,x;
Moves to (y,x) in *win* and gets a character

int mvwgetstr(y,x,str)
WINDOW *win;
int y,x;
char *str;
Moves to (y,x) in *win* and gets a string

int mvwin(win,y,x)
WINDOW *win;
int y,x;
Moves upper corner of *win* to (y,x)

int mvwinch(win,y,x)
WINDOW *win;
int y,x;

Moves to (y,x) in *win* and gets character at current co-ordinates

int mvwinsch(win,y,x,c)
WINDOW *win;
int y,x;
char c;

Moves to (y,x) in *win* and inserts character

WINDOW *newwin(lines,cols,begin_y,begin_x)
int lines, cols, bigin_y, begin_x;

Creates a new window

int nl()

Sets newline mapping

int nocrmode()

Unsets cbreak mode

int noecho()

Unsets echo mode

int nonl()

Unsets newline mapping

int noraw()

Unsets raw mode

int overlay(win1,win2)
WINDOW *win1, *win2;

Overlays *win1* on *win2*

int overwrite(win1,win2)
WINDOW *win1, *win2;

Overwrites *win1* on top of *win2*

int printw(fmt,arg1,arg2,...)
char *fmt;

Prints args on **stdscr**

int raw()

Sets raw mode

int refresh()

Makes current screen look like **stdscr**

int resetty()

Resets tty flags to stored value

int savetty()

Stored current tty flags

int scanw(fmt,arg1,arg2,...)
char *fmt;

Scans for args through **stdscr**

int scroll(win)
WINDOW *win;

Scrolls *win* one line

int scrollok(win,state)
WINDOW *win;
bool state;

Sets scroll flag

int setterm(name)
char *name;

Sets term variables for name

int standend()

Clears standout mode of **stdscr**

int standout()

Sets standout mode for characters in subsequent output to **stdscr**

WINDOW *subwin(win,lines,cols,begin_y,begin_x)
WINDOW *win;
int lines, cols, begin_y, begin_x;

Creates a subwindow in *win*

int touchwin(win)
WINDOW *win;

Prepares *win* for complete update on next refresh.

int unctrl(ch)
char ch;

Printable version of *ch*

int waddch(win,ch)
WINDOW *win;
char ch;

Adds char to *win*

int waddstr(win,str)
WINDOW *win;
char *str;

Adds string to *win*

int wclear(win)
WINDOW *win;
Clear *win*

int wclrtobot(win)
WINDOW *win;
Clears to bottom of *win*

int wclrtoeol(win)
WINDOW *win;
Clears to end of line on *win*

int wdelch(win)
WINDOW *win;
Deletes current character from *win*

int wdeletein(win)
WINDOW *win;
Deletes line from *win*

int werase(win)
WINDOW *win;
Erase *win*

int wgetch(win)
WINDOW *win;
Gets a char through *win*

int wgetstr(win,str)
WINDOW *win;
char *str;
Gets a string through *win*

int winch(win)
WINDOW *win;
Gets char at current (y,x) in *win*

int winsch(win,c)
WINDOW *win;
char c;
Inserts character c in *win*

int winsertln(win)
WINDOW *win;
Inserts a blank line in *win*

int wmove(win,y,x)
WINDOW *win;
int y,x;
Sets current (y,x) co-ordinates on

```
int wprintw(win,fmt,arg1,arg2,...)
WINDOW *win;
char *fmt;
```
Print args on *win*

```
int wrefresh(win)
WINDOW *win;
```
Makes screen look like *win*

```
int wscanw(win,fmt,arg1,arg2,...)
WINDOW *win;
char *fmt;
```
Scans for args through *win*

```
int wstandend(win)
WINDOW *win;
```
Clears standout mode for *win*

```
int wstandout(win)
WINDOW *win;
```
Sets standout mode for characters on subsequent output to *win*

See Also

termcap(M), stty(C), setenv(S), terminfo(S)
XENIX *C Library Guide*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Name

cuserid – Gets the login name of the user.

Syntax

#include <stdio.h>

char *cuserid (s)
char *s;

Description

cuserid returns a pointer to string which represents the login name of the owner of the current process. If (int)*s* is zero, this representation is generated in an internal static area, the address of which is returned. If (int)*s* is nonzero, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The manifest constant **L_cuserid** is defined in **<stdio.h>**.

Diagnostics

If the login name cannot be found, *cuserid* returns NULL; if *s* is nonzero in this case, \0 will be placed at **s*.

See Also

getlogin(S), *getpwent* in getpwent(S)

Notes

cuserid uses *getpwnam* (see *getpwent*(S)); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.

Name

dbminit, fetch, store, delete, firstkey, nextkey – Performs database functions.

Syntax

typedef struct { char *dptr; int dsize; } datum;

int dbminit(file)
char *file;

datum fetch(key)
datum key;

int store(key, content)
datum key, content;

int delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;

Description

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **-ldbm**.

keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length **.dir** and **.pag** files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *firstkey* will return the first key in the database.

With any key *nextkey* will return the next key in the database. This code will traverse the database:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

Diagnostics

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

Notes

The .pag file will contain holes so that its apparent size is about four times its actual content. Older XENIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means *(cp, cat, tp, tar, ar)* without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 512 bytes). Moreover all key/content pairs that hash together must fit on a single block. *store* will return an error in the event that a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function.

These routines are not reentrant, so they should not be used on more than one database at a time.

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Name

defopen, defread – Reads default entries.

Syntax

int defopen(filename)
char *filename;

char *defread(pattern)
char *pattern;

Description

defopen and *defread* are a pair of routines designed to allow easy access to default definition files. XENIX is normally distributed in binary form; the use of default files allows OEMs or site administrators to customize utility defaults without having the source code.

defopen opens the default file named by the pathname in *filename*. *defopen* returns null if it is successful in opening the file, or the *fopen* failure code (*errno*) if the open fails.

defread reads the previously opened file from the beginning until it encounters a line beginning with *pattern*. *defread* then returns a pointer to the first character in the line after the initial *pattern*. If a trailing newline character is read it is replaced by a null byte.

When all items of interest have been extracted from the opened file the program may call *defopen* with the name of another file to be searched, or it may call *defopen* with NULL, which closes the default file without opening another.

Files

The XENIX convention is for a system program *xyz* to store its defaults (if any) in the file **/etc/default/xyz**.

Diagnostics

defopen returns zero on success and nonzero if the open fails. The return value is the *errno* value set by *fopen* (S).

defread returns NULL if a default file is not open, if the indicated pattern could not be found, or if it encounters any line in the file greater than the maximum length of 128 characters.

Notes

The return value points to static data, whose contents are overwritten by each call.

Name

dial – Establishes an out-going terminal line connection.

Syntax

```
#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 it is 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 for use only with an out-going 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 receives 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* is 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 does not have to be specified as it will be determined from the **L-devices** file.

The *telno* element is a pointer to a character string representing the telephone number to be dialed. Such numbers may consist of symbols only described on the *acu*(7). The termination symbol will be supplied by the *dial* function, and shonld not be included in the *telno* string passed to *dial* in the CALL structure.

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

alarm(S), dial(M), read(S), termcap(M), uucp(C), write(S)

Diagnostics

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices listed below 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     /* 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 */

Notes

An *alarm*(S) 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*(C) 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*(S) or *write*(S) 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.

Warnings

When you include the **<dial.h>** header file, the **<termio.h>** header file is automatically included.

Note that the above routine uses **<stdio.h>**, which causes it to increase its program size, otherwise not using standard I/O, more than might be expected.

Name

opendir, readdir, telldir, seekdir, rewinddir, closedir – Performs directory operations.

Syntax

#include <sys/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

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 **NULL** pointer is returned if *filename* cannot be accessed or if it is not a directory.

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 associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* 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 reopened, 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, a d the structure associated with the DIR pointer to be freed.

Sample code which searches a directory for the entry "name" is shown below:

```
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;
```

See Also

close(S), lseek(S), open(S), read(S)

Notes

This routine must be linked with the linker option **-lx**.

Name

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – Generates uniformly distributed pseudo-random numbers.

Syntax

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];

See Also

rand(S)

Description

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

The 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 is $m = 2^{48}$; thus, 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(S)

Notes

These routines are coded in portable C. The source code for the portable version can even be used on computers which do not support floating-point arithmetic. In such a situation, functions *drand48* and *erand48* do not exist; instead, they are replaced by two new functions shown below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval $[0, m-1]$.

Name

dup, dup2 – Duplicates an open file descriptor.

Syntax

int dup (fildes)
int fildes;

int dup2(fildes, fildes2)
int fildes, fildes2;

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).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl* (S).

dup returns the lowest available file descriptor. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

dup will fail if one or more of the following are true:

fildes is not a valid open file descriptor. [EBADF]

Sixty file descriptors are currently open. [EMFILE]

Return Value

Upon successful completion a nonnegative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

Notes

This routine must be linked using the linker option **−lx**.

See Also

creat(S), close(S), exec(S), fcntl(S), open(S), pipe(S)

Name

ecvt, fcvt, gcvt – Performs output conversions.

Syntax

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;

Description

ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string. 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). If the sign of the result is negative, the word pointed to by *sign* is nonzero, otherwise it is zero. The low-order digit is rounded.

fcvt is identical to *ecvt,* except that the correct digit has been rounded for FORTRAN F format output of the number of digits specified by *ndigits*.

gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

See Also

printf(S)

Notes

The return values point to static data whose content is overwritten by each call.

Name

end, etext, edata – Last locations in program.

Syntax

extern char *end;
extern char *etext;
extern char *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* is the first address above the initialized data region. *end* is the first address above the uninitialized data region.

See Also

brk(S), malloc(S).

Warning

No assumptions should be made with respect to the ordering of the program text, initialized data, and uninitialized data regions. For example, the assumption can't be made that the addresses following the address of etext will reference the uninitialized data region.

No assumptions can be made concerning the contiguity of information within a region. A region may be split among different parts of memory. Therefore, no assurance can be made that addresses within a region are consecutive.

Name

erf, erfc – Error function and complementary error function.

Syntax

#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(S)

Notes

These routines must be linked by using the **–lm** linker option.

Name

execl, execv, execle, execve, execlp, execvp – Executes a file.

Syntax

int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp [ ];

int execve (path, argv, envp);
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];

Description

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the "new process file." There can be no return from a successful *exec* because the calling process is overlaid by the new process.

path points to a pathname that identifies the new process file.

file points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(M)). The environment is supplied by the shell (see *sh*(C)).

arg0, arg1, ..., argn are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present, and it must 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 process. 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 for the new process. *Envp* is terminated by a null pointer.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(S). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(S).

If the set-user-ID mode bit of the new process file is set (see *chmod*(S)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

Profiling is disabled for the new process; see *profil*(S).

The new process also inherits the following attributes from the calling process:

Nice value (see *nice*(S))

Process ID

Parent process ID

Process group ID

semadj values (see *semop*(S))

TTY group ID (see *exit*(S) and *signal*(S))

Trace flag (see *ptrace*(S) request 0)

Time left until an alarm clock signal (see *alarm*(S))

Current working directory

Root directory

File mode creation mask (see *umask*(S))

File size limit (see *ulimit*(S))

utime, *stime*, *cutime*, and *cstime* (see *times*(S))

From C, two interfaces are available: *execl* and *execv*. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments. The first argument is conventionally the same as the filename (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance. The arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

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.

argv is directly usable in another *execv* because *argv*[*argc*] is 0.

envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(C) passes an environment entry for each global shell variable defined when the program is called. See *environ*(M) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current program. The *exec* routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, arg1, . . . , argn, 0, environ);
execve(file, argv, environ);
```

execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's pathname do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The new process file has the appropriate access permission, but has an invalid magic number in its header or some other executable file format inconsistency. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is physically available for user programs or the program would not fit on the swap disk. [ENOMEM]

The number of bytes in the new process' argument list is greater than the system-imposed limit of 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

path, *argv*, or *envp* point to an illegal address. [EFAULT]

Return Value

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

See Also

exit(S), fork(S), proctl(S), semop(S)

Notes

exec may still fail when physical memory is larger than the swap disk (see ENOMEM above). However, this restriction may be lifted using one of the following *procti*(S) calls:

PRHUGEX
: Allows programs to be executed by this process even if they exceed the available swap disk space. Such programs must still fit in the available physical memory and the caller's effective user ID must be the super-user. Such HUGE processes are locked in memory to prevent them from being swapped.

PRNORMX
: Makes a process unable to *exec* HUGE programs. This call may be executed by any user.

Name

execseg – Makes a data region executable.

Syntax

#include <xdata.h>

excode_t execseg(*oldaddr, size*)
exdata_t *oldaddr*;
unsigned *size*;

int unexecseg(*addr*)
excode_t *addr*;

Description

execseg(S) is passed the current data address and size of the region to be executed and it returns the starting address of a region that is at least *size* number of bytes which can safely be branched to. On the Intel 8086 and 80286, processor an alias CS descriptor is associated with the same memory as the data segment in which the *oldaddr* region lies. This means that offsets in the executable segment to access a given byte are essentially the same as the offsets in the original data segment, except the selector is different.

Note that "excode_t" and "exdata_t" are 'far' pointers on the 8086 and 80286. On an architecture where pages in the same 'segment' are any combination of read/write/execute, the returned address is identical to the parameter passed to *execseg*(S).

We recommend that programs using this function on 8086- and 80286-based processors be large model, or that programmers be very familiar with "hybrid model" as well as with the use and misuse of far data.

When an error occurs, *execseg*(S) returns ((excode_t)-1), with *errno* set to ENONEM. Errors include an invalid data address or *size*, and an inability to allocate a new data selector.

The *unexecseg*() system call disables an *addr* previously returned from *execseg*(S) from being used as an executable region. Specifically, on the 8086 and 80286 architectures, this call frees the selector used for the executable region. It returns 0 on success, or a -1 on error. For example, if *addr* is not an address returned by *execseg*(S), then a -1 is returned and it cn be used as an executable region.

Example:

```
excode_t funcp; char far *datap;
        .
        .
        .
datap=brkctl(BR_NEWSEG,1000L,0L);
load_with_code(datap,1000)        /*loads executable code into
                                    data region datap*/
funcp=execseg(datap,1000); (*funcp)()
/*call subroutine*/ if (unexecseg (funcp)==-1){
     printf("unexecseg failed\n"); exit(1); }
```

Notes

On the Intel 8086 and 80286 architectures, *execseg*(S) expects far addresses to be passed. Only experienced programmers should use this feature.

Since the *execseg* return value and address arguments are '**far**' pointers, any program including xdata.h must be compiled using the **-Me** option.

The following restrictions apply to the execute data system call. Even though an address and size are passed to *execseg*, the entire segment containing the requested addresses are aliased. The address and size are validated before the aliasing is allowed. No part of the data segment that is aliased may be deallocated (via *sbrk*(S) or *brkctl*(S)) while it is aliased. This restriction applies to the entire segment that is aliased, even if only a small piece of the segment was aliased. After *unexecseg*ing the aliased segment, the data segment may be deallocated. Each call to *execseg* results in a new alias segment being used, even if the data segment is already aliased.

Due to compiler confusion, you may get the message "at least one void operand" when using *execseg*. Please ignore it.

Name

exit, _exit – Terminates a process.

Syntax

exit (status)
void int status;
void _exit (status)
int status;

Description

exit terminates the calling process. 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' termination and the low-order 8 bits (i.e., hits 0377) of *status* are made available to it; see *wait*(S). If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes *wait*(S).

If the parent process of the calling process is not executing a *wait*, 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. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*(S).

The parent process ID of all of the calling process' existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(S)) 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.

For each semaphore for which the calling process has set a *semadj* value (see *semop*(S)), that *semadj* value is added to the *semval* of the specified semaphore.

If the process has a text, data lock, or process, an *unlock* is performed (see *plock*(S)).

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(S).

If the process ID, TTY group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each of the processes that has a process group ID equal to that of the calling process.

The C function *exit* may cause cleanup actions before the process exits. The *_exit* circumvents all cleanup.

See Also

acct(S), intro(S), plock(S), semop(S), signal(S), wait(S)

Warning

See *Warning* in *signal*(S)

Name

exp, log, pow, sqrt, log10 – Performs exponential, logarithm, power, square root functions.

Syntax

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;

double log10 (x)
double x;

Description

exp returns the exponential function of x.

log returns the natural logarithm of x.

pow returns x^y.

sqrt returns the square root of x.

See Also

intro(S), hypot(S), sinh(S)

Diagnostics

exp and *pow* return a HUGE value when the correct value would overflow. An unusually large argument may also result in *errno* being set to ERANGE. *log* and *log10* return HUGE negative values and set *errno* to EDOM when x is nonpositive. A message indicating DOMAIN error (or SING error when x is 0) is printed on the standard error output. *pow* returns a huge negative value and sets *errno* to EDOM when x is nonpositive and y is not an integer, or when x and y are both zero. *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* (S).

Notes

These routines must be linked by using the **-lm** linker option.

Name

fclose, fflush – Closes or flushes a stream.

Syntax

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

Description

fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

fclose is performed automatically upon calling *exit*(S).

fflush causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

These functions return 0 for success, and EOF if any errors were detected.

See Also

close(S), fopen(S), setbuf(S)

Name

fcntl – Controls open files.

Syntax

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;

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. *arg* is either an *int* or a *pointer* , depending on the *cmd* given. See below.

The *cmd*s available are:

F_DUPFD
: Returns a new file descriptor as follows:

  Lowest numbered available file descriptor greater than or equal to *arg*.

  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 sbare the same file status flags).

  The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(S) system calls.

F_GETFD
: Gets 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*, otherwise the file will be closed upon execution of *exec*.

F_SETFD
: Sets the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or otherwise as above).

F_GETFL
: Gets *file* status flags: O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY, or O_APPEND.

F_SETFL Sets *file* status flags to *arg*. Only certain flags can be set.

F_GETLK
: Gets the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg* (see below). 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, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK
: Sets or clears a file segment lock according to the variable of type *struct flock* pointed to by *arg* (see below). The F_SETLK command 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 immediately return an error value of - 1.

F_SETLKW
: This command 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 may 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*), process ID (*l_pid*) and system ID (*l_sysid*) of the segment of the file to be affected as shown below:

```
struct flock {
      short   l_type: /* F_RDLCK, F_WRLCK, F_UNLCK*/
      short   l_whence: /* flag to choose starting offset */
      long    l_start: /* relative offset in bytes */
      long    l_len: /* if 0 then until EOF */
      short   l_pid: /* returned with F_GETLK */
      short   l_sysid: /* returned with F_GETLK */
};
```

l_whence is 0,1 or 2 to indicate that the relative offset will be measured from the start of the file, current position or end of the file, respectively.

The process ID and system ID fields are only used with the F_GETLK command to return the value for a blocking 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 affect. 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*(S) system call.

fcntl fails if one or more of the following is true:

fildes is not a valid open file descriptor. [EBADF]

cmd is F_DUPFD and 60 file descriptors are currently open. [EMFILE]

cmd is F_DUPFD and *arg* is negative or greater than 60. [EINVAL]

cmd is F_GETLK, F_SETLK, or F_SETLKW and *arg* or the data it points to is not valid. [EINVAL]

cmd is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) or write (F_WRLCK) lock and the segment of a file to be locked is by another process or the type is a write lock and the segment of a file to be locked is already read or write locked by another process. [EAGAIN]

cmd is F_SETLK or F_SETLKW, the type of lock is a read or write lock and there are no more file locks available (too many segments are locked). [ENOLOCK]

cmd is F_SETLK, the lock is blocked by a lock from another process and putting the calling process to sleep or waiting for that lock to become free, would cause a deadlock. [EDEADLK] or [EDEADLOCK]

Return Value

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD
A new file descriptor

F_GETFD
Value of flag (only the low-order bit is defined)

F_SETFD
Value other than −1

F_GETFL
Value of file flags

F_SETFL Value other than −1

F_GETLK
Value other than −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.

See Also

close(S), exec(S), lockf(S), open(S)

Notes

fcntl provides mandatory record locking.

Name

ferror, feof, clearerr, fileno – Determines stream status.

Syntax

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream

clearerr (stream)
FILE *stream

int fileno(stream)
FILE *stream;
```

Description

feof returns nonzero when end-of-file is read on the named input *stream*, otherwise zero.

ferror returns nonzero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

clearerr resets the error indication on the named *stream*.

fileno returns the integer file descriptor associated with the *stream*, see *open*(S).

feof, *ferror*, and *fileno* are implemented as macros; they cannot be redeclared.

See Also

open(S), fopen(S)

Name

floor, fabs, ceil, fmod – Performs absolute value, floor, ceiling and remainder functions.

Syntax

#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

fabs returns $|x|$.

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 number f such that $x = iy + f$, for some integer i, and $0 \leq f < y$.

See Also

abs(S)

Notes

These routines must be linked by using the **–lm** linker option.

Name

fopen, freopen, fdopen – Opens a stream.

Syntax

#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

Description

fopen opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

type is a character string having one of the following values:

r Open for reading

w Create for writing

a Append; open for writing at end of file, or create for writing

r+ Open for update (reading and writing)

w+ 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*. It returns the original value of *stream*. The original stream is closed, regardless of whether the open call ultimately succeeds.

freopen is typically used to attach the preopened constant names **stdin, stdout,** and **stderr** to specified files.

fdopen associates a stream with a file descriptor obtained from *open*, *dup*, *creat*, or *pipe*(S). The *type* of the stream must agree with the mode of the open file. The *type* must be provided because the standard I/O library has no way to query the type of an open file descriptor. *fdopen* returns the new stream.

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 the end of the file.

When a file is opened for append (that is, 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.

See Also

open(S), fclose(S)

Diagnostics

fopen and *freopen* return the pointer NULL if *filename* cannot be accessed.

Name

fork – Creates a new process.

Syntax

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*(S))

signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, function address)

set-user-ID mode bit

set-group-ID mode bit

process group ID

tty group ID (see *exit*(S) and *signal*(S))

current working directory

root directory

file mode creation mask (see *umask*(S))

file size limit (see *ulimit*(S))

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*(S)).

The child process' *utime*, *stime*, *cutime*, and *cstime* are set to 0; see *times*(S).

The time left on the parent's alarm clock is not passed on to the child.

fork returns a value of 0 to the child process.

fork returns the process ID of the child process to the parent process.

fork will fail and no child process will be created if one or more of the following are true:

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. [EAGAIN]

Not enough memory is available to create the forked image. [ENOMEM]

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.

See Also

exec(S), sdget(S), semop(S), shmop(S), wait(S)

Name

fread, fwrite – Performs buffered binary input and output.

Syntax

#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 reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* 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*. It returns the number of items actually read.

fwrite appends at most *nitems* of data of the type of **ptr* beginning at *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*. *fwrite* increments the file pointer in *stream*, if defined, by the number of bytes written. It returns the number of items actually written.

See Also

fopen(S), getc(S), gets(S), printf(S), putc(S), puts(S), read(S), scanf(S), write(S)

Diagnostics

fread and *fwrite* return the number of items read or written. If *sizeof* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

Name

frexp, ldexp, modf – Splits floating-point number into a mantissa and an exponent.

Syntax

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 $x * 2^{n}$ wher 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 *exptr*. If *value* is 0, both results returned by *frexp* are 0.

ldexp returns the quantity *value**(2***exp*).

modf returns the positive fractional part of *value* and stores the integer part indirectly through *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.

Notes

These routines must be linked by using the **-lm** linker option.

Name

fseek, ftell, rewind – Repositions a file pointer in a stream.

Syntax

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

void rewind(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, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

fseek undoes any effects of *ungetc*(S).

After *fseek* or *rewind*, the next operation on an update file may be either input or output.

ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. The offset is measured in bytes.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

See Also

lseek(S), fopen(S), popen(S), ungetc(S)

Diagnostics

fseek returns nonzero for improper seeks, otherwise zero.

Name

ftw – Walks a file tree.

Syntax

#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;

Description

ftw recursively descends the directory hierarchy routed in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string. This string contains the name of the object, a pointer to a *stat* structure with information about the object, and an integer. Possible values for the integer include 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 be successfully executed. These values are defined in the **<ftw.h>** header file. If the integer is FTW_DNR, descendants of the directory will not be processed. If the integer is FTW_NS, the *stat* structure will contain meaningless information. For example, a file in a directory with read but without execute permission could cause FTW_FN to be passed to *fn*.

ftw visits a directory before visiting any of its descendants. The file tree traversal continues until the tree is exhausted, *fn* returns a nonzero value, or some error is detected within *ftw* (for example, an I/O error). If the file tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops traversing the file tree and returns the value 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. *depth* limits the number of file descriptors. This argument must not be greater than the number of file descriptors currently available for use. Zero or negative values for *depth* are interpreted as 1. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

See Also

stat(S), malloc(S)

Notes

Because *ftw* is recursive, it can terminate with a memory fault when applied to very deep file structures.

ftw uses *malloc*(S) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated (for example, by *longjmp* being executed by *fn* or by an interrupt routine), *ftw* will not have a chance to free that storage, and it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and have *fn* return a nonzero value at its next invocation.

Name

gamma – Performs log gamma function.

Syntax

#include <math.h>
extern int signgam;

double gamma (x)
double x;

Description

gamma returns *ln* $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program fragment might be used to calculate Γ:

```
if((y = gamma (x)) >LN_MAXDOUBLE)
        error ();
y = exp (y) * signgam;
```

where LN_MAXDOUBLE is the least value that causes *exp*(S) to return a range error and is defined in the **<values.h>** header file.

Diagnostics

For negative integer arguments, a HUGE value 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 a HUGE value and *errno* is set to ERANGE.

These error-handling procedures may be changed with the *matherr*(S) function.

See Also

exp(S), matherr(S)

Notes

These routines must be linked by using the **−lm** linker option.

Name

getc, getchar, fgetc, getw – Gets character or word from a stream.

Syntax

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

Description

getc and *getchar* are macros. *getc* returns the next character from the named input *stream* as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar()* is identical to *getc*(*stdin*).

fgetc behaves like *getc*, but is a genuine function, not a macro; it may therefore be used as an argument. *fgetc* runs more slowly than *getc*, but takes less space per invocation.

getw returns the next word 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 same as an integer and varies from machine to machine. *getw* assumes no special alignment in the file.

See Also

ferror(S), fopen(S), fread(S), gets(S), putc(S), scanf(S)

Diagnostics

These functions return the integer constant EOF at the end-of-file or upon a read error. Because EOF is a valid integer, *ferror*(S) should be used to detect *getw* errors.

Notes

stream arguments with side effects are treated incorrectly because *getc* is implemented as a macro. In particular, "getc(*f++)" doesn't work properly. *fgetc* should be used instead.

Files written using *putw*(S) are machine-dependent and may not be read using *getw* on a different processor because of possible differences in word length and byte ordering.

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.

Name

getcwd – Get the pathname of current working directory.

Syntax

char *getcwd (pnbuf, maxlen)
char *pnbuf;
int maxlen;

Description

getcwd returns a pointer to the current directory pathname. If *pnbuf* is a NULL pointer, *getcwd* will obtain *maxlen* bytes of space using *malloc*(S). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*(S). If *pnbuf* is not a NULL pointer, then the pathname is placed in the space pointed to by *pnbuf* and *pnbuf* is returned.

In all cases, the value of *maxlen* must be at least two greater than the length of the pathname to be returned.

getcwd is implemented by using *popen*(S) to pipe the output of the *pwd*(C) command into the specified string space.

Example

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

See Also

pwd(C), malloc(S), popen(S)

Errors

[EINVAL] *size* is zero

[ENOMEM] no space is available

[ERANGE] *size* not large enough to hold the path name.

Diagnostics

Returns NULL with *errno* set if *maxlen* is not large enough.

Notes

maxlen must be 2 more than the true length of the pathname.

Name

getenv – Gets value for environment name.

Syntax

char *getenv (name)
char *name;

Description

getenv searches the environment list (see *environ*(M)) for a string of the form *name*=*value* and returns pointer to the *value* if such a string is present. Otherwise a NULL pointer is returned.

See Also

sh(C), exec(S)

Name

getgrent, getgrgid, getgrnam, setgrent, endgrent – Get group file entry.

Syntax

#include <grp.h>

struct group *getgrent ( );

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

int setgrent ( );

int endgrent ( );

Description

getgrent, *getgrgid* and *getgrnam* each return pointers. The format of the structure is defined in **/usr/include/grp.h.**

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

getgrent reads the next line of the file, so successive calls may be used to search the entire file. *getgrgid* and *getgrnam* search from the beginning of the file until a matching *gid* or *name* is found, or end-of-file is encountered.

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.

Files

/etc/group

See Also

getlogin(S), getpwent(S), group(M)

Diagnostics

A null pointer (0) is returned on end-of-file or error.

Notes

All information is contained in a static area, so it must be copied if it is to be saved.

Name

getlogin – Gets login name.

Syntax

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 device, it returns NULL. 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

cuserid(S), getgrent(S), getpwent(S), utmp(M)

Diagnostics

Returns NULL if name not found.

Notes

The return values point to static data whose content is overwritten by each call.

Name

getopt – Gets option letter from argument vector.

Syntax

```
#include <stdio.h>

int getopt (argc, argv, optstring)
int argc;
char *argv[ ];
char *optstring;
extern char *optarg;
extern int optind, opterr;
```

Description

getopt returns the next option letter in *argv* 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 whitespace. *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. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first nonoption 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.

Examples

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 int optind;
        extern char *optarg;
        .
        .
        .
        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;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (S);
        }
        for( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
        .
        .
        .
}
```

Name

getpass – Reads a password.

Syntax

char *getpass (prompt)
char *prompt;

Description

getpass reads a password from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most eight characters.

Files

/dev/tty

Notes

The return value points to static data whose content is overwritten by each call.

Name

getpid, getpgrp, getppid – Gets process, process group, and parent process IDs.

Syntax

int getpid ()

int getpgrp ()

int getppid ()

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.

See Also

exec(S), fork(S), intro(S), setpgrp(S), signal(S)

Name

getpw – Gets password for a given user ID.

Syntax

int getpw (uid, buf)
int uid;
char *buf;

Description

getpw searches the password file for the *uid*, and fills in *buf* with the corresponding line; it returns nonzero if *uid* could not be found. The line is null-terminated. *uid* must be an integer value.

Files

/etc/passwd

See Also

getpwent(S), passwd(M)

Diagnostics

Returns nonzero on error.

Notes

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(S) for routines to use instead.

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent – Gets password file entry.

Syntax

#include <pwd.h>

struct passwd *getpwent ( );

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

int setpwent ( );

int endpwent ( );

Description

getpwent, *getpwuid* and *getpwnam* each returns a pointer to a structure containing the fields of an entry line in the password file. The structure of a password entry is defined in **/usr/include/pwd.h.**

The fields have meanings described in *passwd*(M). (The *pw_comment* field is unused.)

getpwent reads the next line in the file, so successive calls can be used to search the entire file. *getpwuid* and *getpwnam* search from the beginning of the file until a matching *uid* or *name* is found, or EOF is encountered.

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.

Files

/etc/passwd

See Also

getlogin(S), getgrent(S), passwd(M)

Diagnostics

Null pointer (0) returned on EOF or error.

Notes

All information is contained in a static area so it must be copied if it is to be saved.

Name

gets, fgets – Gets a string from a stream.

Syntax

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

Description

gets reads a string into *s* from the standard input stream stdin. The function replaces the newline character at the end of the string with a null character before copying to *s*. *gets* returns a pointer to *s*.

fgets reads characters from the *stream* until a newline character is encountered or until *n*–1 characters have been read. The characters are then copied to the string *s*. A null character is automatically appended to the end of the string before copying. *fgets* returns a pointer to *s*.

See Also

ferror(S), fopen(S), fread(S), getc(S), puts(S), scanf(S)

Diagnostics

gets and *fgets* return the constant pointer NULL upon end-of-file or error.

Notes

gets deletes the newline ending its input, but *fgets* keeps it.

Name

getuid, geteuid, getgid, getegid – Gets real user, effective user, real group, and effective group IDs.

Syntax

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

intro(S), setuid(S)

Name

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – Accesses utmp file entry.

Syntax

```
#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 the following type of structure:

```
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;           /*The exit status of a process*/
        } ut_exit;                  /*The exit status of a process*/
                                    /*marked as DEAD_PROCESS.*/
        time_t  ut_time;            /*Time entry was made*/
};
```

getutent reads the next entry from a **utmp**-like file. If the file is not already open, *getutent* opens it; when *getutent* 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, then *getutid* returns a pointer to the first entry whose type matches one of these four types and whose *ut_id* matches *id* -> *ut_id*. If the end of the file is reached without a match, *getutid* fails.

getutline searches forward from the current point in the **utmp** file until it reaches an entry of the type LOGIN_PROCESS or USER_PROCESS which has an *ut_line* string matching the *line* -> *ut_line* string. If the end of the file is reached without a match, *getutline* fails.

pututline writes out the supplied *utmp* structure into the **utmp** file. If *pututline* finds that it is not already in the proper place in the file, it uses *getutid* to search forward for the proper place. A user of *pututline* could search for the proper place using one of the *getut* routines. If *pututline* does not find a matching slot for the new entry, it adds a new entry to the end of the file.

setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if the user desires that the entire file be examined.

endutent closes the currently opened file.

utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. Generally, this other file will be **/etc/wtmp**. If this file does not exist, it 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 open and saves the new file name.

Files

/etc/utmp
/etc/wtmp

See Also

ttyslot(S), utmp(M)

Diagnostics

A NULL pointer is returned upon failure to read (either because of permissions or the end of the file) or upon failure to write.

Comments

With these routines, the most current entry is saved in a static structure. Multiple accesses require that the structure 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 the routine is searching for, the search stops. For this reason, to use *getutline* to search for multiple occurrences, the user must to remove the static after each success, or *getutline* will just return the same pointer over and over again.

There is one exception to the rule of removing the structure before further reads are done: the implicit read done by *pututline* (in cases where it finds that it is not already in the correct place in the file) will not hurt the contents of the static structure returned by *getutent*, *getutid*, or *getutline* routines if the user has just modified those contents and passed the pointer back to *pututline*.

These routines used 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.

Name

hsearch, hcreate, hdestroy – Manages hash search tables.

Syntax

#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. This routine 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

item.data points to any other data associated with the comparison 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 the appropriate point. FIND indicates that no entry should be made. The return of a NULL pointer indicates unsuccessful resolution.

hcreate makes sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the highest number of entries the table will contain. The algorithm can adjust this number upwards in order to obtain mathematically favorable circumstances.

hdestroy destroys the search table, and may be followed by another call to *hcreate*.

hsearch uses open addressing with a multiplicative hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV
: Use the remainder modulo table size as the hash function instead of the multiplicative algorithm.

USCR
: Use a User Supplied Comparison Routine for determining table membership. The routine should be named *hcompar* and should behave in a manner similar to *strcmp* (see *string*(S)).

CHAINED
: Use a linked list to resolve collisions. If this option is selected, the user has the following options:

START	Place new entries at the beginning of the linked list (default is at the end).
SORTUP	Keep the linked list sorted by key in ascending order.
SORTDOWN	Keep the linked list sorted by key in descending order.

In addition, there are preprocessor flags for obtaining debugging printout (**-DDEBUG**) and for including a test driver in the calling routine (**-DDRIVER**). Consult the source code for further details.

Return Value

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.

Example

The following fragment of code 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 (scan("%s%d%d", str_ptr, &infor_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(S), lsearch(S), malloc(S), string(S), tsearch(S).

Diagnostics

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.

Notes

Only one hash search table may be active at any given time.

Warning

hsearch and *hcreate* use *malloc*(S) to allocate space.

Name

hypot, cabs – Determines Euclidean distance.

Syntax

```
#include <math.h>

double hypot (x, y)
double x, y;

double cabs (z)
struct {double x, y;} z;
```

Description

hypot and *cabs* return:

sqrt(x*x + y*y)

Both take precautions against unwarranted overflows.

See Also

sqrt in exp(S), matherr(S)

Diagnostics

When the correct value reaches overflow, *hypot* returns a HUGE value and sets *errno* to ERANGE.

These error-handling procedures may be changed with the *matherr*(S) function.

Notes

These routines must be linked by using the **-lm** linker option.

Name

ioctl – Controls character devices.

Syntax

#include <sys/ioctl.h>

int ioctl(fildes, request, arg)
int fildes;

Description

ioctl performs a variety of functions on character special files (devices). The arguments *request* and *arg* depend on which device *ioctl* is being applied to. The writeups of various devices in Section M discuss how *ioctl* applies to them.

ioctl fails if one or more of the following are true:

A signal is caught during *ioctl* system call. [EINTR]

fildes is not a valid open file descriptor. [EBADF]

fildes is not associated with a character special device. [ENOTTY]

request or *arg* is not valid. See *termio* (M). [EINVAL]

A signal was caught during the *ioctl* system call. [EINTR]

Return Value

If an error has occurred, a value of –1 is returned and *errno* is set to indicate the error.

See Also

tty(M), termio(M)

Name

kill – Sends a signal to a process or a group of processes.

Syntax

#include <signal.h>

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*(S), 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 effective user ID of the receiving process unless, the effective user ID of the sending process is super-user, or the process is sending to itself.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro* (S)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* 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 *proc0* and *proc1* whose real user ID is equal to the effective user 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 *proc0* and *proc1*.

If *pid* is negative but not –1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

Sig is not a valid signal number. [EINVAL]

No process can be found corresponding to that specified by *pid*. [ESRCH]

The sending process is not sending to itself, its effective user ID is not super-user, and its effective user ID does not match the real user ID of the receiving process. [EPERM]

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

kill(C), getpid(S), setpgrp(S), signal(S)

Name

l3tol, ltol3 – Converts between 3-byte integers and long integers.

Syntax

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* 3-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 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

See Also

filesystem(F)

Name

link – Links a new filename to an existing file.

Syntax

int link (path1, path2)
char *path1, *path2;

Description

path1 points to a pathname naming an existing file. *path2* points to a pathname giving the new filename to be linked. *link* makes a new link by creating a new directory entry for the existing file using the new name. The contents of the existing file can then be accessed using either name.

link will fail and no link will be created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* already exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not super-user. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

path2 points to a null pathname. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

path points outside the process' allocated address space. [EFAULT]

The maximum number of lines to a file is exceeded. [EMLINK]

The directory to contain the file cannot be extended. [ENOSPC]

Return Value

When the linking procedure is successfully completed, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

See Also

ln(C), unlink(S)

Name

lock – Locks a process in primary memory.

Syntax

int lock(flag);
int flag;

Description

If the *flag* argument is nonzero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is un*lock*ed. This call may only be executed by the super-user.

Notes

*lock*ed processes interfere with the compaction of primary memory and can cause deadlock. Systems with small memory configurations should avoid using this call. It is best to lock process soon after booting because that will tend to lock them into one end of memory.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine must be linked using the linker option **-lx**.

Name

lockf – Provide semaphores and record locking on files.

Syntax

#include <unistd.h>

int lockf(fildes, function, size)
long size;
int fildes, function;

Description

lockf locks a specified region of the file given by the file descriptor, *fildes*, against access by all other processes. Other processes which attempt to use the locked region will either return an error or wait until the region is unlocked. More than one region in a file can be locked. When the process closes the file (or terminates), all locks are removed. See *fcntl*(S) for more information about record locking.

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish a lock with the *lockf* function call.

The *function* argument specifies what action to take. The possible values are defined in **<unistd.h>** and as follows:

F_ULOCK
: Unlock a previously locked region.

F_LOCK
: Lock the region for exclusive use. If the region is not available, the calling process sleeps until the region is available.

F_TLOCK
: Test for locks, then lock the region for exclusive use. If the region is not available, *lockf* returns immediately and sets *errno* to EAGAIN .

F_TEST
: Test the region for other processes' locks. This argument is used to determine whether or not another process has placed a lock on the specified region.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current position 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 the *size* is 0, the region extends from the current position in the file to the current or future end of the file. An area does not need to be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked region for the same process. When this occurs, or if overlapping regions occur, the regions are combined. If the request requires that a new element be added to the table of active locks and this table is already full, an [EDEADLK] (or [EDEADLOCK]) 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. F_TLOCK will cause the function to return a -1 and set *errno* to [EAGAIN] error if the region is already locked by another process.

F_ULOCK requests may, in whole or in 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 region of a locked region requires an additional element in the table of active locks. If this table is full, an [EDEADLK] (or [EDEADLOCK]) error is returned and the requested region is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Therefore, calls to *lockf*(S) or *fcntl*(S) scan for a deadlock prior to sleeping on a locked resource. An [EDEADLK] (or [EDEADLOCK]) error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(S) routine may be used to provide a timeout facility in applications that require this facility.

The *lockf* routine will fail if one or more of the following are true:

> *fildes* is not a valid open descriptor. [EBADF]
>
> *cmd* is F_TLOCK or F_TEST and the region is already locked by another process. [EAGAIN]
>
> *cmd* is F_LOCK or F_TLOCK and a deadlock occurs. Also the *cmd* is either of the above or F_U OCK, and there are not enough entries in the system lock table to honor the request. [EDEADLK] or [EDEADLOCK]

Return Values

When the lock routine is successfully completed, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

alarm(S), chmod(S), close(S), creat(S), fcntl(S), open(S), read(S), write(S),

Notes

Record and file locking should not be used in combination with the standard I/O routines, such as *fopen*(S), *fread*(S), and *fwrite*(S). Instead, the more primitive, non-buffered routines such as *open*(S) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is or was locked.

Name

locking – Locks or unlocks a file region for reading or writing.

Syntax

#include <sys/types.h>
#include <sys/locking.h>

int locking(fildes, mode, size);
int fildes, mode;
long size;

Description

locking allows a specified number of bytes in a file to be controlled by the locking process. Other processes which attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked depending upon the mode in which the file region was locked.

A file must be open with read or read/write permission for a read lock to be performed. Write or read/write permission is required for a write lock. If either of these conditions are not met, the lock will fail with the error EINVAL.

A process that attempts to write to or read a file region that has been locked against reading and writing by another process (using the LK_LOCK or LK_NBLCK mode) will sleep until the region of the file has been released by the locking process.

A process that attempts to write to a file region that has been locked against writing by another process (using the LK_RLCK or LK_NBRLCK mode) will sleep until the region of the file has been released by the locking process, but a read request for that file region will proceed normally.

A process that attempts to lock a region of a file that contains areas that have been locked by other processes will sleep if it has specified the LK_LOCK or LK_RLCK mode in its lock request, but will return with the error EACCES if it specified LK_NBLCK or LK_NBRLCK.

fildes is the value returned from a successful *creat, open, dup,* or *pipe* system call.

mode specifies the type of lock operation to be performed on the file region. The available values for mode are:

LK_UNLCK 0
: Unlocks the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK 1
: Locks the specified region. The calling process will sleep until the entire region is available if any part of it has been locked by a different process. The region is then locked for the calling process and no other process may read or write in any part of the locked region. (lock against read and write).

LK_NBLCK 2
: Locks the specified region. If any part of the region is already locked by a different process, return the error EACCES instead of waiting for the region to become available for locking (non-blocking lockrequest).

LK_RLCK 3
: Same as LK_LOCK except that the locked region may be read by other processes (read permitted lock).

LK_NBRLCK 4
: Same as LK_NBLCK except that the locked region may be read by other processes (nonblocking, read permitted lock).

The *locking* utility uses the current file pointer position as the starting point for the *locking* of the file segment. So a typical sequence of commands to *lock* a specific range within a file might be as follows:

```
fd=open("datafile",O_RDWR);
lseek(fd, 200L, 0);
locking(fd, LK_LOCK, 200L);
```

Accordingly, to *lock* or *unlock* an entire file a *seek* to the beginning of the file (position 0) must be done and then a *locking* call must be executed with a size of 0.

size is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current offset in the file. If *size* is 0, the entire file (up to a maximum of 2 to the power of 30 bytes) is locked or unlocked. *size* may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by size.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process' locked area. Thus calls to *locking*, *read*, or *write* scan for a deadlock prior to sleeping on a locked region. An EDEADLK (or EDEADLOCK) error return is made if sleeping on the locked region would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs, or when adjacent regions are locked, the regions are combined into a single area if the mode of the lock is the same (i.e.; either read permitted or regular lock). If the mode of the overlapping locks differ, the locked areas will be assigned assuming that the *most recent request* must be satisfied. Thus if a read only lock is applied to a region, or part of a region, that had been previously locked by the same process against both reading and writing, the area of the file specified by the new lock will be locked for read only, while the remaining region, if any, will remain locked against reading and writing. There is no arbitrary limit to the number of regions which may be locked in a file. There is however a system-wide limit on the total number of locked regions. This limit is 200 for XENIX systems.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional locked element to hold the separated section. If the lock table is full, an error is returned, and the requested region is not released. Only the process which locked the file region may unlock it. An unlock request for a region that the process does not have locked, or that is already unlocked, has no effect. When a process terminates, all locked regions controlled by that process are unlocked.

If a process has done more than one open on a file, *all* locks put on the file by that process will be released on the first close of the file.

Although no error is returned if locks are applied to special files or pipes, read/write operations on these types of files will ignore the locks. Locks may not be applied to a directory.

See Also

creat(S), open(S), read(S), write(S), dup(S), close(S), lseek(S)

Diagnostics

locking returns the value (int) -1 if an error occurs. If any portion of the region has been locked by another process for the LK_LOCK and LK_RLCK actions and the lock request is to test only, *errno* is set to EAGAIN when used with XENIX System V binaries. If the binary using this routine is a XENIX 3.0 binary, this *errno* is set to EACCES. If the file specified is a directory, *errno* is set to EACCES. If locking the region would cause a deadlock, *errno* is set to EDEADLK (or EDEADLOCK). If there are no more free internal locks, *errno* is set to EDEADLK (or EDEADLOCK).

Notes

This routine must be linked with the linker option **-lx**.

Name

logname – Finds login name of user.

Syntax

char *logname();

Description

logname returns the current user name from *login* to stdout.

Files

/etc/profile

See Also

env(C), login(M), profile(M), environ(M)

Name

lsearch, lfind – Performs linear search and update.

Syntax

```
#include <stdio.h>
#include <search.h>
char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar)();

char *lfind (key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar) ();
```

Description

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm Q. It returns a pointer into a table indicating the location at which a datum may be found. If the item does not occur, it is added at the end of the table. The first argument is a pointer to the datum to be located in the table. The second argument is a pointer to the base of the table. The third argument is the address of an integer containing the number of items in the table. It is incremented if the item is added to the table. The fourth argument is the width of an element in bytes. The last argument is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return zero if the items are equal, and nonzero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table.

Example

This fragment of code will read ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates:

```
#include <stdio.h>
#include <search.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(S), hsearch(S), qsort(S), tsearch(S)

Diagnostics

If the datum searched for 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.

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

Unpredictable events can occur if there is not enough room in the table to add a new item.

Name

lseek – Moves read/write file pointer.

Syntax

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.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

> *fildes* is not an open file descriptor. [EBADF]
>
> *fildes* is associated with a pipe or fifo. [ESPIPE]
>
> *whence* is not 0, 1 or 2. [EINVAL and SIGSYS signal]
>
> The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return Value

Upon successful completion, a nonnegative integer indicating the file pointer value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), fcntl(S), open(S)

Name

malloc, free, realloc, calloc – Allocates main memory.

Syntax

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

There are two versions of the *malloc(S)* package. Both versions are documented in these *malloc*(S) manual pages; the description for the other package starts on page 3. This portion of the manual page documents the standard, default *malloc*(S) package. This version of *malloc* and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to *free* is a pointer to a block previously allocated by *malloc*; tbis space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first 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 *sbrk*(S)) 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

brkctl(S), malloc(S), sbrk(S)

Diagnostics

malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the area has been detectably corrupted by storing outside the bounds of a block. When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

Note

As noted, *malloc* calls *sbrk* to allocate memory. Since *sbrk* takes a signed integer as its argument, *malloc* will fail if an attempt is made to allocate more memory than a signed integer will hold (32K -1).

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 and more flexible implementation see the *malloc*(S) documented on pages 3-5 of this manual entry.

Name

malloc, free, realloc, calloc, mallopt, mallinfo – Allocates main memory quickly.

Syntax

```
#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
```

Description

There are two versions of the *malloc(S)* package. This is the library version which provides a simple general-purpose memory allocation package, that runs considerably faster than the other *malloc*(S) package. Both versions are documented in these *malloc*(S) manual pages; the description of the standard default package starts on page 1.

This *malloc*(S) package is found in the library "malloc" and is loaded when the option **-hnalloc** is used with *cc*(CP) or *ld*(CP).

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 destroyed (see *mallopt* below for a way to change this behavior).

Undefined results 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 0.

M_NLBLKS
: Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *numlblks* must be greater than 0. 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. 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

XENIX Programmer's Guide
brkctl(S), malloc(S), sbrk(S)

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 or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

Warnings

This package usually uses more data space than the other *malloc*(S).

The code size is also bigger than the other *malloc*(S).

Note that unlike the other *malloc*(S), 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 the other *malloc*(S) have not been duplicated.

These routines must be linked with the **-lmalloc** linker option.

Name

matherr – Error-handling function.

Syntax

#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 0 ** non-pos	M, 0	-	-	-	-	-
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.

Notes

These routines must be linked by using the **-lm** linker option.

Name

memccpy, memchr, memcmp, memcpy, memset - Memory operations.

Syntax

```
#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; however, they do not check for the overflow of any receiving memory area. Memory areas are arrays of characters bounded by a count, not terminated by a null character.

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*. If *c* was not found in the first *n* characters of *s2*, *memccpy* returns a NULL pointer.

memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*. If *c* does not occur, this function returns a NULL pointer.

memcmp compares its arguments, looking at the first *n* characters only, and returns an integer. This integer will be less than, equal to, or greater than 0 according to whether *s1* is lexicographically less than, equal to, or greater than *s2*.

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*.

These routines are declared in the **<memory.h>** header file.

Notes

memcmp uses native character comparison, which is signed on some systems and unsigned on others; therefore, the sign of the value returned is device-dependent when one of the characters has its high-order bit set.

Character movement is performed differently in different implementations, so overlapping moves may yield unexpected results.

Name

mknod – Makes a directory, or a special or ordinary file.

Syntax

int mknod (path, mode, dev)
char *path;
int mode, dev;

Description

mknod creates a new file named by the pathname 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 Named pipe special
- 0020000 Character special
- 0040000 Directory
- 0050000 Name special file
- 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 file's owner ID is set to the process' effective user ID. The file's group ID is set to the process' effective group ID.

The low-order 9 bits of *mode* are modified by the process' file mode creation mask: all bits set in the process' file mode creation mask are cleared. See *umask*(S). If *mode* indicates a block, character, or name special file, then *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block, character, or name special file, then *dev* is ignored. For block and character special files, *dev* is the special file's device number. For name special files, *dev* is the type of the name file, either a shared memory file or a semaphore.

mknod may be invoked only by the super-user for file types other than named pipe-special files.

mknod will fail and the new file will not be created if one or more of the following are true:

The process' effective user ID is not super-user. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

path points outside the process' allocated address space. [EFAULT]

The directory to contain the new file cannot be extended. [ENOSPC]

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

chmod(S), creatsem(S), exec(S), filesystem(F), mkdir(C), mknod(C), sdget(S), umask(S),

Notes

Semaphore files should be created with the *creatsem*(S) system call.

Share data files should be created with the *sdget*(S) system call.

Name

mktemp – Makes a unique filename.

Syntax

char *mktemp(template)
char *template;

Description

mktemp replaces *template* with a unique filename and returns the address of template. The template should look like a filename with six trailing **X**'s, which will be replaced with the current process ID preceded by a letter. The letter will be chosen so that the resulting name does not duplicate an existing file.

See Also

getpid(S), tmpfile(S), tmpnam(S)

Notes

It is possible to run out of letters.

Name

monitor – Prepares execution profile.

Syntax

void monitor (lowpc,highpc,buffer,bufsize,nfunc)
int (*lowpc) ( ), (*highpc) ( );
short *buffer;
int bufsize, nfunc;

Description

monitor is an interface to *profil*(S). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a user-supplied array of *bufsize* short integers. *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*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of *cc*(CP) are recorded. For the 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, etext, buf, bufsize, nfunc);
```

etext lies just above all the program text.

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor((int (*)())0);
```

prof(CP) can then be used to examine the results.

Files

mon.out

See Also

cc(CP), prof(CP), profil(S)

Notes

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Warning

Profiling gives incorrect results for hybrid model 286 programs (i.e. those with 16 bit text pointers within modules and 32 bit text pointers between modules).

Name

mount – Mounts a file system.

Syntax

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

Description

mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to pathnames.

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.

mount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

dir is not a directory. [ENOTDIR]

spec or *dir* points outside the process' allocated address space. [EFAULT]

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. [EBUSY]

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

mount(C), umount(S)

Name

msgctl – Provides message control operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msquid_ds *buf;

Description

msgctl provides for message control operations specified by *cmd*.

The *cmds* available are:

IPC_STAT
: Places the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. Contents of this structure are defined in *intro*(S).

IPC_SET
: Sets the value of the following members of the data structure associated with *msqid* into the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mod /* only low 9 bits*/
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either a super-user or to the value of *msg_perm.uid* in the data structure associated with *msqid*. Only a super-user can raise the value of *msg_qbytes*.

IPC_RMID
: Removes the message queue identifier specified by *msqid* from the system and destroys 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 a super-user or to the value of *msg_perm.uid* in the data structure associated with *msqid*.

msgctl will fail if one or more of the following are true:

msqid is not a valid message queue identifier. [EINVAL]

cmd is not a valid command. [EINVAL]

cmd is equal to *IPC_STAT* and *buf* points to an address in read-only shared data. [EINVAL]

cmd is equal to *IPC_STAT* and read operation permission is denied to the calling process (see *intro* (S)). [EACCES]

cmd is equal to *IPC_RMID* or *IPC_SET*. The effective user ID of the calling process does not equal that of a super-user nor does it equal the value of *msg_perm.uid* 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.

buf points to an illegal address. [EFAULT]

Return Value

A value of 0 is returned upon successful completion. Otherwise, −1 is returned and *errno* is set to indicate the error.

See Also

intro(S), msgget(S), msgop(S)

Notes

Programs using this function must be compiled with the -Me compiler option.

Name

msgget – Gets message queue.

Syntax

#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, an associated message queue, and data structure (see *intro*(S)) are created for *key* if one of the following is 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".

Values for the data structure associated with the new message queue identifier are initialized as follows:

msg_perm.cuid and **msg_perm.uid** are set equal to the effective user ID of the calling process. **msg_perm.cgid** and **msg_perm.gid** are set equal to the effective group ID 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**, 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.

msgget fails if one or more of the following is true:

A message queue identifier exists for *key*; however, operation permission as specified by the low-order 9 bits of *msgflg* would not be granted (see *intro*(S)). [EACCES]

A message queue identifier does not exist for *key* and (*msgflg* & IPC_CREAT) is "false". [ENOENT]

A message queue identifier would be created but the system-imposed limit on the maximum number of allowed message queue identifiers for the system would be exceeded. [ENOSPC]

A message queue identifier exists for the *key* but ((*msgflg* & IPC_CREAT) & (*msgflg* && IPC_EXCL)) is "true". [EEXIST]

Return Value

Upon successful completion, the message queue identifier is returned. This is a non-negative integer. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

See Also

intro(S), msgctl(S), msgop(S).

Notes

Programs using this function must be compiled with the -Me compiler option.

Name

msgop – Message operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msquid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Description

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

msgp points to the structure containing the message. The structure contains 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 text of length *msgsz* bytes. *msgsz* can range from 0 to a maximum imposed by the system.

msgflg specifies the action to be taken if one or more of the following conditions is true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *intro*(S)).

The number of messages on all the queues system-wide equals the system-imposed limit.

The actions *msgflg* specifies include:

The message will not be sent and the calling process will return immediately if (*msgflg* & IPC_NOWAIT) is true.

If (*msgflg* & IPC_NOWAIT) is false, the calling process will suspend execution until one of following the occurs:

The condition causing the suspension no longer exists. In this case, the message is sent.

msqid is removed from the system (see *msgctl*(S)). In this case, *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 described in *signal*(S).

msgsnd will fail and no message will be sent if one or more of the following are true:

msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(S)). [EACCES]

mtype is less than 1. [EINVAL]

The message cannot be sent for one of the preceding reasons and (*msgflg* & IPC_NOWAIT) is true. [EAGAIN]

msgsz is less than zero or greater than the system-imposed limit. [EINVAL]

msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see *Intro*(S)).

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 reads a message from the queue associated with the message queue identifier (*msqid*) and places it in the structure pointed to by *msgp*. The structure contains the following members:

```
long    mtype;          /* message type */
char    mtext[];        /* message text */
```

mtype is the received message's type. This is specified by the sending process. *mtext* is the text of the message. *msgsz* gives the size in bytes of *mtext*. If the received message is larger than *msgsz* bytes and (*msgflg* & MSG_NOERROR) is true, the message is truncated to *msgsz* bytes. The truncated part of the message is lost and no notice of the truncation is given to the calling process.

msgtyp specifies the type of message requested:

If *msgtype* equals zero, the first message on the queue is received.

If *msgtyp* is greater than zero, the first message of type *msgtyp* is received.

If *msgtyp* is less than zero, the first message of the lowest type less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies an action if a message of the desired type is not on the queue. These include:

If (*msgflg* & *IPC_NOWAIT*) is true, calling process returns immediately with a return value of −1 and *errno* is set equal to ENOMSG.

If (*msgflg* & *IPC_NOWAIT*) is false, calling process suspends execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. *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 described in *signal* (S).

msgrcv will fail and no message will be received if one or more of the following are true:

msqid is not a valid message queue identifier. [EINVAL] *buf* points to an address in read-only shared data. [EINVAL]

Operation permission is denied to the calling process. [EACCES]

msgsz is less than 0. [EINVAL]

mtext is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is false. [E2BIG]

The queue does not contain a message of the desired type and (*msgtyp* & IPC_NOWAIT) is true. [ENOMSG]

msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken on the data structure associated with *msqid* (see *Intro* (S)).

msg_qnum is decreased 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 because of a signal received, a value of −1 is returned to the calling process and *errno* is set to EINTR. If these operations return because *msqid* was removed from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return values are:

msgsnd returns 0.

msgrcv returns a value equal to the number of bytes placed into *mtext*.

Otherwise, −1 is returned and *errno* is set to indicate the error.

See Also

intro(S), msgctl(S), msgget(S), signal(S).

Notes

Programs using this function must be compiled with the -Me compiler option.

Name

nap – Suspends execution for a short interval.

Syntax

long nap(period)
long period;

Description

The current process is suspended from execution for at least the number of milliseconds specified by *period*, or until a signal is received.

Return Value

On successful completion, a long integer indicating the number of milliseconds actually slept is returned. If the process recieved a signal while napping, the return value will be -1, and *errno* will be set to EINTR.

See Also

sleep(S)

Notes

This function is driven by the system clock, which in most cases has a granularity of tens of milliseconds. This function must be linked with the linker option **-lx**.

Name

nice – Changes priority of a process.

Syntax

int nice (incr)
int incr;

Description

nice adds the value of *incr* to the nice value of the calling process. A process' nice value is a positive number for which a higher 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.

nice will not change the nice value if *incr* is negative or greater than 40, and if the effective user ID of the calling process is not super-user. [EPERM]

Return Value

Upon successful completion, *nice* returns the new nice value minus 20. Note that *nice* is unusual in the way return codes are handled. It differs from most other system calls in two ways: the value –1 is a valid return code (in the case where the new nice value is 19), and the system call either works or ignores the request; there is never an error.

See Also

exec(S), nice(C)

Name

nlist – Gets entries from name list.

Syntax

#include <a.out.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl

Description

nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(F) for a discussion of the symbol table structure.

See Also

a.out(F), xlist(S)

Diagnostics

nlist return −1 and sets all type entries to 0 if the file cannot be read, is not an object file, or contains an invalid name list. Otherwise, *nlist* returns 0. A return value of 0 does not indicate that any or all symbols were found.

Name

open – Opens file for reading or writing.

Syntax

#include <fcntl.h>
int open (path, oflag[, mode])
char *path;
int oflag, mode;

Description

path points to a pathname naming a file. *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 using flags from the following list (only one of the first three flags below may be used):

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* (S) and *write* (S).

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 file's owner ID is set to the process' effective user ID, the file's group ID is set to the process' effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(S)):

All bits set in the process' file mode creation mask are cleared. See *umask*(S).

The "save text image after execution bit" of the mode is cleared. See *chmod*(S).

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_SYNCW Every write to this file descriptor will be synchronous, that is, when the *write* system call completes, data is guaranteed to have been written to disk.

Upon successful completion, a nonnegative integer, the file descriptor, is returned.

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*(S).

No process may have more than 60 file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Sixty file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

path points outside the process' allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

A signal was caught during the *open* system call. [EINTR]

The system file table is full. [ENFILE]

The directory to contain the file cannot be extended, the file does not exist, and O_CREAT is specified. [ENOSPC]

Return Value

Upon successful completion, a nonnegative integer, namely a file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

See Also

chmod(S), close(S), creat(S), dup(S), fcntl(S), lseek(S), read(S), umask(S), write(S)

Notes

The O_SYNCHW flag is a XENIX specific enhancement which may not be present in all UNIX implementations.

Name

opensem – Opens a semaphore.

Syntax

int opensem(sem_name)
char *sem_name;

sem_num = opensem(sem_name);

Description

opensem opens a semaphore named by *sem_name* and returns the unique semaphore identification number *sem_num* used by *waitsem* and *sigsem*. *creatsem* should always be called to initialize the semaphore before the first attempt to open it.

System Compatibility

opensem can only be used to open semaphores created under XENIX version 3.0, not for XENIX System V semaphores.

See Also

creatsem(S), sigsem(S), waitsem(S)

Diagnostics

opensem returns a value of −1 if an error occurs. If the semaphore named does not exist, *errno* is set to ENOENT. If the file specified is not a semaphore file (i.e., a file previously created by a process using a call to *creatsem*), *errno* is set to ENOTNAM. If the semaphore has become invalid due to inappropriate use, *errno* is set to ENAVAIL.

Notes

This feature is a XENIX specific enhancement which may not be present in all UNIX implementations. This function must be linked with the linker option **-lx**.

Warning

It is not advisable to open the same semaphore more than once. Although it is possible to do this, it may result in a serious deadlock.

Name

pause – Suspends a process until a signal occurs.

Syntax

int 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 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*(S)), the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and *errno* set to EINTR.

See Also

alarm(S), kill(S), signal(S), wait(S)

Name

perror, sys_errlist, sys_nerr, errno – Sends system error messages.

Syntax

void perror(s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

Description

perror produces a short error message on the standard error, describing the last error encountered during a system call from a C program. First the argument string *s* is printed, then a colon, then the message and a newline. To be of most use, the argument string should be 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 correct calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *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.

See Also

intro(S)

Name

pipe – Creates an interprocess pipe.

Syntax

int pipe (fildes)
int fildes[2];

Description

pipe creates an I/O mechanism called a pipe and returns two file descriptors in the array *fildes*. *fildes*[0] is opened for reading and *fildes*[1] is opened for writing and the O_NDELAY flag is clear. The descriptors remain open across *fork*(S) system calls, making communication between parent and child possible.

Writes up to 10240 bytes of data (10 times BSIZE) are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 60 file descriptors open simultaneously.

pipe will fail if 19 or more file descriptors are currently open. [EMFILE] It will also fail if the system file table is full. [ENFILE]

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

sh(C), read(S), write(S), fork(S), popen(S)

Name

plock – Lock process, text, or data in memory.

Syntax

#include <sys/lock.h>
int plock (op)
int op;

Description

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be root user to use this call. *op* specifies the following:

PROCLOCK
Lock text and data segments into memory.

TXTLOCK
Lock text segment into memory.

DATLOCK
Lock data segment into memory.

UNLOCK
Remove all process locks.

plock will fail and not perform the requested operation if one or more of the following are true:

The effective user ID of the calling process is not root. [EPERM]

op is equal to PROLOCK 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]

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.

See Also

exec(S), exit(S), fork(S)

Name

popen, pclose – Initiates I/O to or from a process.

Syntax

```
#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 process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

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 between processes, a type "r" command may be used as an input filter, and a type "w" as an output filter.

See Also

pipe(S), wait(S), fclose(S), fopen(S), system(S)

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.

Notes

Only one stream opened by *popen* can be in use at once. Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing; see *fclose*(S).

Name

printf, fprintf, sprintf – Formats output.

Syntax

#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 placed (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 *arg*s 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 *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s 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 is preceded with a "0" (e.g., %04), the converted value will be padded with zeroes. If the width is preceded with a blank (e.g., % 4), the value will be preceded with blanks. Padding with zeroes may be applied to numeric conversions only. Strings and characters cannot be zero padded.

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** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*.

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 *arg*s 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 prepended 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 **e**, **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** (**X**) conversion, a nonzero result will have **0x** (**0X**) prepended 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 (**d**), unsigned octal (**o**), unsigned decimal (**u**), 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. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is **o**, **x**, or **X** *and* the # flag is present).

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±dd", 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, 6 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 exactly two digits. However, if the value to be printed is greater than or equal to 1E+100, additional exponent digits will be pointed as necessary.

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.

% Print a %; no argument is converted.

In no case does a nonexistent 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 *putchar* had been called (see *putc*(S)).

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, %.2d:%.2d", weekday, month, day,
 hour, min);
```

To print π to five decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

See Also

ecvt(S), putc(S), scanf(S)

Name

proctl - Controls active processes or process groups.

Syntax

```
#include <sys/proctl.h>

proctl(pid, command, arg)
int pid, command;
char *arg;
```

Description

proctl performs a variety of functions on active processes or process groups. It has the same form as the *ioctl*(S) system call, except that a process ID (*pid*) is substituted for a file descriptor as the first parameter.

command is an integer mnemonic, specifying the action to be taken, and *arg* is a pointer to a data structure which defines the parameters associated with the *command* if necessary.

If *pid* is greater than zero (0), the *command* affects the process whose process ID is equal to *pid*. *pid* may be 1.

If *pid* is zero, the command is sent to all processes, except processes 0 and 1 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 the super-user, the command is sent to all processes, except processes 0 and 1 whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super-user, the command is sent to all processes except processes 0 and 1.

If *pid* is negative but not -1, a signal is sent to all processes whose process group ID is equal to the absolute value of *pid*.

proctl will fail if one or more of the following are true:

command or *arg* is not valid. [EINVAL]

No process can be found to match the specified *pid*. [ESRCH]

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. [EPERM]

The program has requested more memory than is available. [ENOMEM]

arg is not a valid address. [EFAULT]

Memory Restrictions

exec(S) may fail when the required physical memory is larger than the available swap space. This restriction may be lifted using one of the following *proctl* commands:

PRHUGEX

Allows programs to be executed by this process even if they exceed the available swap space. Such programs must still fit in the available physical memory and the caller's effective user ID must be super-user. Such HUGE processes are locked in memory to prevent them from being swapped. Processes that are marked HUGE with this system call but are not greater than the size of the swapper behave normally but can expand into a HUGE, locked process.

PRNORMX

Makes a process unable to *exec*(S) HUGE programs. This call may be executed by any user. If an attempt is made to classify a process as normal using the PRNORMX call when the process is already too big to swap, the *proctl* call will fail, returning EINVAL.

For example, you can use the following code to allow a process to be executed even if it exceeds the available memory swapping space:

```
if (argc < 2) {
            fputs ("usage: runbig command arg ...\n", stderr);
            exit(2);
}
argv[argc] = 0;

if (proctl(getpid(), PRHUGEX, (char *) 0) < 0) {
            perror ("runbig");
            exit(1);
}
```

Return Value

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

See Also

exec(S), ioctl(S), kill(S)

Notes

This function must be linked with the linker option **-lx.**

Name

profil – Creates an execution time profile.

Syntax

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, scale;
int (*offset) ();

Description

buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter is examined each clock tick, where a clock tick is some fraction of a second given in *machine*(HW). *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented. An "entry" is defined as a series of bytes with length sizeof(short).

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.

See Also

prof(CP), monitor(S)

Name

ptrace – Traces a process.

Syntax

int ptrace (request, pid, addr, data);
int request, pid, data, addr;

Description

ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is in the implementation of breakpoint debugging; see *adb* (CP). The child process behaves normally until it encounters a signal (see *signal* (S) for the list), at which time it enters a stopped state and its parent is notified via *wait* (S). When the child is in the stopped state, its parent can examine and modify its "memory 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 *addr* argument is dependant on the underlying machine type, specifically the process memory model. On systems where the memory management mechanism provides a uniform and linear address space to user processes, the argument is declared as:

```
int *addr;
```

which is sufficient to address any location in the process' memory. On machines where the user address space is segmented (even if the particular program being traced has only one segment allocated), the form of the *addr* argument is:

```
struct saddr {
        unsigned short  sa_seg;
        long            sa_off;
} *addr;
```

which allows the caller to specify segment and offset in the process address space.

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*(S). 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 The word at location *addr* in the address space of the child is returned to the parent process. If I and 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. 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*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. 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 a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. 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 follow:

–The general registers

–Any floating-point status registers

–Certain bits of the processor status

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. In a linear address space memory model, the value of *addr* must be (int *)1, or in a segmented address space the segment part of *addr* must be zero and the offset part of *addr* must be (int *)1. 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*(S).

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. This is part of the mechanism for implementing breakpoints. The exact implementation and behaviour is somewhat CPU dependant.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* system call is used to determine when a process stops; in such a case the termination status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To prevent security violations, *ptrace* inhibits the set-user-id facility on subsequent *exec*(S) 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:

request is an illegal number. [EIO]

pid identifies a child that does not exist or has not executed a *ptrace* with request 0. [ESRCH]

Notes

The implementation and precise behaviour of this system call is inherently tied to the specific CPU and process memory model in use on a particular machine. Code using this call is likely to not be portable across all implementations without some change.

See Also

adb(CP), exec(S), signal(S), wait(S), machine(HW)

Name

putc, putchar, fputc, putw – Puts a character or word on a stream.

Syntax

```
#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 appends the character *c* to the named output *stream* (at the position where the file pointer, if defined, is pointing). It returns the character written.

putchar(*c*) is defined as *putc* (*c*, **stdout**).

fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *fputc* runs more slowly than *putc*, but takes less space per invocation.

putw appends the word (i.e., integer) *w* to the output *stream*. *putw* neither assumes nor causes special alignment in the file.

The standard stream **stdout** is normally buffered if and only if the output does not refer to a terminal; this default may be changed by *setbuf*(S). The standard stream **stderr** is by default unbuffered unconditionally, but use of *freopen* (see *fopen*(S)) causes it to become buffered or line-buffered; *setbuf*(S), again, sets the state to whatever is desired. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. See *fflush* in *fclose*(S).

See Also

fclose(S), ferror(S), fopen(S), fread(S), getc(S), printf(S), puts(S)

Diagnostics

When a character or word is successfully put on a stream, these functions each return the value they have written. These functions return the constant EOF upon error. 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*(S) should be used to detect *putw* errors.

Notes

The *stream* argument with side effects is not treated correctly, because *putc* is implemented as a macro. In particular,

```
putc (c, *f++);
```

does not 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 – Changes or adds value to environment.

Syntax

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*.

See Also

environ(M), exec(S), getenv(S), malloc(S)

Diagnostics

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

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*(S) 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 – Writes a password file entry.

Syntax

#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;

Description

putpwent is the inverse of *getpwent*(S). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*. The line matches the format of **/etc/passwd**.

See Also

passwd(M), getpwent(S)

Diagnostics

putpwent returns nonzero if an error was detected during its operation, otherwise zero.

Name

puts, fputs – Puts a string on a stream.

Syntax

#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;

Description

puts copies the null-terminated string *s* to the standard output stream **stdout** and appends a newline character.

fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminating null character.

Diagnostics

Both routines return EOF on error.

See Also

ferror(S), fopen(S), fread(S), gets(S), printf(S), putc(S)

Notes

puts appends a newline, *fputs* does not.

Name

qsort – Performs a quicker sort.

Syntax

void qsort (base, nel, width, compar)
char *base;
unsigned nel, width;
int (*compar)( );

Description

qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according to how much the first argument is to be considered less than, equal to, or greater than the second.

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

bsearch(S), lsearch(S), sort(C), string(S)

Name

rand, srand – Generates a random number.

Syntax

void srand (seed)
unsigned seed;

int rand ( )

Description

rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

The generator is reinitialized by calling *srand* with **1** as argument. It can be set to a random starting point by calling *srand* with an unsigned integer in argument *seed.*

See Also

drand48(S)

Note

The spectral properties of *rand* are limited. *drand48*(S) provides a much better, more elaborate, random-number generator.

Name

rdchk – Checks to see if there is data to be read.

Syntax

int rdchk(fdes);
int fdes;

Description

rdchk checks to see if a process will block if it attempts to read the file designated by *fdes*. *rdchk* returns 1 if there is data to be read or if it is the end of the file (EOF). In this context, the proper sequence of calls using rdchk is:

```
if(rdchk(fildes) > 0)
        read(fildes, buffer, nbytes);
```

See Also

read(S)

Diagnostics

rdchk returns -1 if an error occurs (e.g., EBADF), 0 if the process will block if it issues a *read* and 1 if it is okay to read. EBADF is returned if a *rdchk* is done on a semaphore file or if the file specified doesn't exist.

Notes

This function must be linked with the linker option **-lx**.

Name

read – Reads from a file.

Syntax

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

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*.

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 file 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 the file is associated with a communication line (see *ioctl*(S) and *tty*(M)), or 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 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 character special file 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.

read will fail if one or more of the following are true:

fildes is not a valid file descriptor open for reading. [EBADF]

buf points outside the allocated address space. [EFAULT]

A signal was caught during the *read* system call. [EINTR]

Return Value

Upon successful completion a nonnegative integer is returned indicating the number of bytes actually read. Otherwise, −1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), fcntl(S), ioctl(S), open(S), pipe(S), rdchk(S), tty(M)

Notes

Reading a region of a file locked with *locking* causes *read* to hang indefinitely until the locked region is unlocked.

Name

regex, regcmp – Compiles and executes regular expressions.

Syntax

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*(S) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *regcmp*(CP) 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 zero 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 derived from the editor, *ed*(C) 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 the newline.

– 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 last or first 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. *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. {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 present, 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 parenthesis. 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 newline 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* (CP)) against *string*.

See Also

ed(C), regcmp(CP), free(S), malloc(S)

Notes

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* (S) reuses the same vector saving time and space:

```
/* user's program */
          . . .
malloc(n)
{
        static int rebuf[256];
        return &rebuf;
}
```

Name

regexp – Regular expression compile and match routines.

Syntax

```
#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 step(string, expbuf)
char *string, *expbuf;
```

Description

This page describes general purpose regular expression matching routines in the form of *ed*(C), defined in **/usr/include/regexp.h**. Programs such as *ed*(C), *sed*(C), *grep*(C), *expr*(C), 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 that the highest address that 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*(C), 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*(C).

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 one, 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 only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the 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 a one 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*(C) and *sed*(C) 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 routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

Examples

The following is an example of how the regular expression macros and calls look from *grep*(C):

```
#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>
...
                  compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
                  if(step(linebuf, expbuf))
                                succeed();
```

Files

/usr/include/regexp.h

See Also

ed(C), grep(C), sed(C).

Notes

The handling of *circf* is awkward.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.

Name

sbrk, brk – Changes data segment space allocation.

Syntax

char *sbrk (incr)
int incr;

int brk (addr)
char *addr;

Description

sbrk and *brk* are used to dynamically change the amount of space allocated for the data segment of the calling process; see *exec*(S). The change is made by resetting the break value of the process. 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.

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.

In 286 large model programs, if *incr* is greater than tbe number of unallocated bytes remaining in the current data segment, *sbrk* automatically allocates all the requested bytes in a new data segment. This guarantees that the requested bytes will reside entirely in one segment. If *incr* is negative and its absolute value is equal to the number of allocated bytes in the current data segment, the segment is automatically freed for other use. If *incr* is negative and its absolute value is greater than the number of allocated bytes in the current segment, the segment is freed, and the additional bytes are removed from the previous data segment. (The previous data segment contains space allocated by the most recent *sbrk* that did not affect the current segment.)

sbrk will fail without making any change in the allocated space if:

> A change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(S)). [ENOMEM]
>
> An attempt is made to remove more space than has actually been allocated.
>
> An attempt to remove space causes the new break value to be less than the original break value. The original break value is always taken to be break value when process execution began

plus any shared data bytes that have been allocated since that time.

brk sets the the current break value to *addr*, and changes the allocated space accordingly. *brk* fails if the address references a data segment that does not exist, or if it references beyond the maximum possible size of the current data segment.

Return Value

Upon successful completion, *sbrk* returns a pointer to the beginning of the allocated space. *brk* returns 0 on successful completion. Otherwise, a value of -1 is returned and *errno* is set to indicate the error. In large model programs, if *sbrk* allocates a new data segment, the return value is the starting address of that new segment.

See Also

exec(S)

Notes

In 286 large model programs, the call "sbrk(0)" does not necessarily return the starting address of the next *sbrk* call. In particular, if the next call causes an additional data segment to be allocated, the break values returned by these two calls will not be the same. The return value from "sbrk(0)" should only be regarded as a marker for the original end of data.

Name

scanf, fscanf, sscanf – Converts and formats input.

Syntax

#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. Blanks, tabs, or newlines which cause input to be read up to the next nonwhitespace 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, and a conversion character.

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 nonspace 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 preceding an input field is ignored.

The conversion character 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 characters are allowed:

% 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.

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 space character or a newline.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace 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.

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 optionally signed integer.

[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 caret (^), 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 caret) 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 capitalized and/or 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 capitalized and/or 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 conflicting character is left unread in the input stream.) This is very important to remember, because subtle errors can occur when not taking this into account.

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; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign 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];
scanf ("%2d%f%*d%[1234567890]", &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*(S)) will return "a".

See Also

atof(S), getc(S), printf(S), strtod(S), strtol(S)

Diagnostics

These functions return EOF on end of input and a short count for missing or illegal data items.

Notes

The success of literal matches and suppressed assigaments is not directly determinable.

Trailing whitespace (including a newline) is left unread unless matched in the control string.

Name

sdenter, sdleave – Synchronizes access to a shared data segment.

Syntax

#include <sys/sd.h>

int sdenter(addr,flags)
char *addr;
int flags;

int sdleave(addr)
char *addr;

Description

sdenter is used to indicate that the current process is about to access the contents of a shared data segment. *addr* is the valid return code from a previous *sdget* (S) call. The actions performed depend on the value of *flags*. *flags* values are formed by OR-ing together entries from the following list:

SD_NOWAIT If another process has called *sdenter* but not *sdleave* for the indicated segment, and the segment was not created with the SD_UNLOCK flag set, return an ENAVAIL error instead of waiting for the segment to become free.

SD_WRITE Indicates that the process wants to write data to the shared data segment. A process that has attached to a shared data segment with the SD_RDONLY flag set will not be allowed to enter with the SD_WRITE flag set.

sdleave is used to indicate that the current process is done modifying the contents of a shared data segment.

Only changes made between invocatations of *sdenter* and *sdleave* are guaranteed to be reflected in other processes. *sdenter* and *sdleave* are very fast; consequently, it is recommended that they be called frequently rather than leave *sdenter* in effect for any period of time. In particular, system calls should be avoided between *sdenter* and *sdleave* calls.

The *fork* system call is forbidden between calls to *sdenter* and *sdleave* if the segment was created without the SD_UNLOCK flag.

Return Value

Successful calls return 0. Unsuccessful calls return -1, and *errno* is set to indicate the error. *errno* is set to EINVAL if a process does an *sdenter* with the SD_WRITE flag set and the segment is already attached with the SD_RDONLY flag set. *errno* is set to ENAVAIL if the SD_NOWAIT flag is set for sdenter call and the shared data segment is not free.

See Also

sdget(S), sdgetv(S)

Notes

This feature is a XENIX specific enhancement and may not be present on all UNIX implementations. This routine must be linked with the linker option **-lx**.

Name

sdget, sdfree – Attaches and detaches a shared data segment.

Syntax

```
#include <sys/sd.h>

char *sdget(path, flags, size, [mode])
char *path;
int flags, mode;
long size;

int sdfree(addr);
char *addr;
```

Description

sdget attaches a shared data segment to the data space of the current process. The actions performed are controlled by the value of *flags*. *flags* values are constructed by OR-ing flags from the following list:

SD_RDONLY
: Attach the segment for reading only.

SD_WRITE
: Attach the segment for both reading and writing.

SD_CREAT
: If the segment named by *path* exists and is not in use (active), this flag will have the same effect as creating a segment from scratch. Otherwise, the segment is created according to the values of *size* and *mode*. Read and write access to the segment is granted to other processes based on the permissions passed in *mode*, and functions the same as those for regular files. Execute permission is meaningless. The segment is initialized to contain all zeroes.

SD_UNLOCK
: If the segment is created because of this call, the segment will be made so that more than one process can be between sdenter and sdleave calls.

sdfree detaches the current process from the shared data segment that is attached at the specified address. If the current process has done *sdenter* but not an *sdleave* for the specified segment, *sdleave* will be done before detaching the segment.

When no process remains attached to the segment, the contents of that segment disappear, and no process can attach to the segment without creating it by using the SD_CREAT flag in *sdget*. *errno* is set to EEXIST if a process tries to create a shared data segment that exists and is in use. *errno* is set to ENOTNAM if a process attempts an *sdget* on a file that exists but is not a shared data type.

Notes

Use of the SD_UNLOCK flag on systems without hardware support for shared data may cause severe performance degradation.

For 286 programs, it is strongly recommended that *sdget* and other shared data functions be reserved for large model programs only. Small or middle model programs that attempt to use shared data may run out of available memory. Also, due to the 286 hardware, it is not possible to enforce the read-only aspect of small model shared data. However, read-only segments are honored in large model programs.

The 386 provides a 32 bit address space, even in small model. As a result, shared data may be conveniently used without regard to the restrictions that apply to 286 programs.

sdget automatically increments the process's original break value to the memory location immediately after the shared data segment. This affects subsequent *sbrk* or *brk* calls which attempt to restore the original break value. In particular, attempts to restore the break value to its value before the *sdget* call causes an error.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine must be linked using the linker option **-lx**.

Return Value

On successful completion, the address at which the segment was attached is returned. Otherwise, -1 is returned, and *errno* is set to indicate the error. *errno* is set to EINVAL if a process does an *sdget* on a shared data segment to which it is already attached. *errno* is set to EEXIST if a process tries to create a shared data segment that exists and is in use. *errno* is set to ENOTNAM if a process attempts an *sdget* on a file that exists but is not a shared data type.

The mode parameter must be included on the first call of the *sdget()* function.

See Also

sdenter(S), sdgetv(S), sbrk(S)

Notes

The *size* variable in *sdget* has changed from unsigned to long between XENIX Version 3.0 and XENIX System V. Although this requires that source code be modified to use a long *size* parameter when compiling with the System V libraries, an unsigned *size* parameter will still be correctly interpreted by the kernel when passed by binaries compiled with the Version 3.0 libraries.

Name

sdgetv, sdwaitv – Synchronizes shared data access.

Syntax

#include <sys/sd.h>

int sdgetv(addr)
int sdwaitv(addr, vnum)
char *addr;
int vnum;

Description

sdgetv and *sdwaitv* may be used to synchronize cooperating processes that are using shared data segments. The return value of both routines is the version number of the shared data segment attached to the process at address *addr*. The version number of a segment changes whenever some process does an *sdleave* for that segment.

sdgetv simply returns the version number of the indicated segment.

sdwaitv forces the current process to sleep until the version number for the indicated segment is no longer equal to *vnum*.

Return Value

Upon successful completion, both *sdgetv* and *sdwaitv* return a positive integer that is the current version number for the indicated shared data segment. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See Also

sdenter(S), sdget(S)

Notes

This routine must be linked using the linker option **-lx**.

Name

semctl – Controls semaphore operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
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 *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL Return the value of semval (see *intro* (S)).

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*. {READ}

GETNCNT Return the value of *semncnt*. {READ}

GETZCNT Return the value of *semzcnt*. {READ}

The following *cmd*s 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 *cmd*s 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 *intro* (S).

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 the super-user or to the value of **sem_perm.uid** 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 the super-user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

semctl will fail if one or more of the following are true:

semid is not a valid semaphore identifier. [EINVAL]

semnum is less than zero or greater than **sem_nsems**. [EINVAL]

cmd is not a valid command. [EINVAL]

cmd is equal to GETALL or IPC_STAT and *arg* points to an address in read-only shared data. [EINVAL]

Operation permission is denied to the calling process (see *intro* (S)). [EACCES]

cmd is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum. [ERANGE]

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 **sem_perm.uid** in the data structure associated with *semid*. [EPERM] *arg.buf* points to an illegal address. [EFAULT]

arg.array points to an illegal address. [EFAULT]

Return Value

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

intro(S), semget(S), semop(S)

Notes

Programs using this function must be compiled with the **-Me** compiler option.

Name

semget – Gets set of semaphores.

Syntax

#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 an associated data structure and set containing *nsems* semaphores (see *intro*(S)) 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.

semget will fail if one or more of the following are true:

nsems is either less than or equal to zero or greater than the system-imposed limit. [EINVAL]

A semaphore identifier exists for *key*, but operation permission (see *intro*(S)) as specified by the low-order 9 bits of *semflg* would not be granted. [EACCES]

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. [EINVAL]

A semaphore identifier does not exist for *key* and (*semflg* & IPC_CREAT) is "false". [ENOENT]

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed system wide semaphore identifiers would be exceeded. [ENOSPC]

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed system wide semaphores would be exceeded. [ENOSPC]

A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT ") and (" *semflg* & IPC_EXCL)) is "true". [EEXIST]

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

intro(S), semctl(S), semop(S)

Notes

Programmers using this function must be compiled with the **-Me** compiler option.

Name

semop – Performs semaphore operations.

Syntax

#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 automatically 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:

```
short    sem_num;    /* semaphore nmnber */
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*.

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* (S)) 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' *semadj* value (see *exit*(S)) 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* ssociated 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' *semadj* value for the specified semaphore.
>
> The *semid* for which the calling process is awaiting action is removed from the system (see *semctl*(S)). 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*(S).

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' *semadj* value for the specified semaphore.

If *sem_op* is zero, one of the following will occur:

If *semval is zero, semop* will return immediately.

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*(S).

semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

semid is not a valid semaphore identifier. [EINVAL]

sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*. [EFBIG]

nsops is greater than the system-imposed maximum. [E2BIG]

Operation permission is denied to the calling process (see *intro*(S)). [EACCES]

The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true". [EAGAIN]

The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded. [ENOSPC]

The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit. [EINVAL]

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. [ERANGE]

sops points to an illegal address. [EFAULT]

Upon successful completion, the value of *semid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

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 value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

See Also

exec(S), exit(S), fork(S), intro(S), semctl(S), semget(S), signal(S)

Notes

If SEMVMX = 32767, *semop* will not be able to make *semval* overflow the limit (ERANGE) because sem_op≥ +32768 (signed short) looks like negative sem_op. Therefore, it will not increase semval to put it over the limit; instead, it will try to subtract ≥ 32768 from semval (EAGAIN). Programs using this function must be compiled with the **-Me** compiler option.

Name

setbuf, setvbuf – Assigns buffering to a stream.

Syntax

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;
int setvbuf (stream, type, buf, size)
FILE *stream;
char *buf;
int type, size;
```

Description

setbuf is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

A manifest constant BUFSIZ, defined in the **<stdio.h>** 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. *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.

A buffer is normally obtained from *malloc*(S) upon the first *getc*(S) or *putc*(S) on the file, except that output streams directed to terminals, and the standard error stream **stderr** are normally not buffered. A common source of error is allocation of buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

See Also

fopen(S), getc(S), malloc(S), putc(S), stdio(S)

Diagnostics

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

Name

setjmp, longjmp – Performs a nonlocal "goto".

Syntax

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

Description

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in *env* for later use by *longjmp*. It returns a value of 0.

longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the corresponding call to *setjmp*. The routine which calls *setjmp* must not itself have returned in the interim. *longjmp* cannot return a value of 0. If *longjmp* is invoked with a second argmnent of 0, it will return a value of 1. All accessible data have values as of the time *longjmp* was called. The only exception to this is register variables. The value of register variables is undefined in the routine that called *setjmp* when the corresponding *longjmp* is invoked.

See Also

signal(S)

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

setpgrp – Sets process group ID.

Syntax

int setpgrp ()

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.

There are many ramifications to be considered before invoking *setpgrp*. When a process is made a process group leader with *setpgrp*, the terminal that controlled the process that issued the *setpgrp* statement is lost as the controlling terminal for the new process group. The new process group takes as its controlling terminal the next terminal it opens that is not already open. All child processes of the new process group leader are controlled by the new controlling terminal.

The controlling terminal is responsible for signals (INTR, KILL, EOF) sent to the process group leader and it child processes. If there is no controlling terminal, it becomes more difficult to interrupt a process.

As an example, *setpgrp* is used to separate *daemon* processes from controlling terminals so that they may not be interrupted from any terminal by a KILL or INTR signal.

Return Value

setpgrp returns the value of the new process group ID.

See Also

exec(S), fork(S), getpid(S), intro(S), kill(S), signal(S), termio(M)

Name

setuid, setgid – Sets user and group IDs.

Syntax

int setuid (uid)
int uid;

int setgid (gid)
int gid;

Description

setuid is used to set the real user ID and effective user ID of the calling process.

setgid is used to set the real group ID and effective group ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

setuid will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

The *uid* is out of range. [EINVAL]

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(S) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*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.

See Also

getuid(S), intro(S)

Name

shmctl – Controls shared memory operations.

Syntax

```
#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 *intro* (S).

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 the super-user or to the value of **shm_perm.uid** 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. This *cmd* can only be executed by a process that has an effective user ID equal to either that of the super-user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

Diagnostics

shmctl will fail if one or more of the following are true:

shmid is not a valid shared memory identifier. [EINVAL]

cmd is not a valid command. [EINVAL]

cmd is equal to **IPC_STAT** and operation permission is denied to the calling process (see *intro*(S)). [EACCES]

cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of the super-user and it is not equal to the value of **shm_perm.uid** in the data structure associated with *shmid*. [EPERM]

buf points to an illegal address. [EFAULT]

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

intro(S), shmget(S), shmop(S)

Notes

Programs using this function must be compiled with **-Me** compiler option.

Name

shmget – Gets a shared memory segment.

Syntax

#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 an associated data structure and shared memory segment of size *size* bytes (see *intro*(S)) 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.

shmget will fail if one or more of the following are true:

size is less than the system-imposed minimum or greater than the system-imposed maximum. The minimum for 286 processes is 1 byte, and the maximum is 64K or 65535 bytes. The minimum and maximum for 386 processes are configurable. [EINVAL]

A shared memory identifier exists for *key* but operation permission (see *intro*(S)) as specified by the low-order 9 bits of *shmflg* would not be granted. [EACCES]

A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size*, which cannot be equal to zero. [EINVAL]

A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false". [ENOENT]

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. [ENOSPC]

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. [ENOMEM]

A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg* & **IPC_EXCL**)) is "true". [EEXIST]

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

intro(S), shmctl(S), shmop(S)

Notes

Programs using this function must be compiled with **-Me** compiler option.

Name

shmop – Performs shared memory operations.

Syntax

For 386 processes:

```
#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;
```

For 286 processes:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char far *shmat (shmid, shmaddr, shmflg)
int shmid;
char far *shmaddr;
int shmflg;

int shmdt (shmaddr)
char far *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. 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.

For 286 processes, if *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "true," the segment is attached at the first available address given by (*shmaddr* – (*shmaddr* modulus SHMLBA)) (SHMLBA = 64K or 65536 bytes).

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*.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true", otherwise it is attached for reading and writing.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*. *shmat* will fail and not attach the shared memory segment if one or more of the following are true:

shmid is not a valid shared memory identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(S)). [EACCES]

The available data space is not large enough to accommodate the shared memory segment. [ENOMEM]

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. [EINVAL]

For 286 processes, the shared memory segment is already attached by the calling process. [EINVAL]

The number of shared memory segments attached to the calling process would exceed the system-imposed limit. [EMFILE]

shmdt detaches the shared memory segment located at the address specified by *shmaddr* from the calling process data segment. [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. [EINVAL]

Return Values

Upon successful completion, the return values are 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(S), exit(S), fork(S), intro(S), shmctl(S), shmget(S)

Notes

Programs using this function must be compiled with the **-Me** compiler option.

For 286 processes, if a program is compiled using small or middle model, the char far variables cannot be used as arguments to the standard **libc.a** routines because these routines require char near pointers. If the **libc.a** routines are required, the program must be compiled using large or huge model. If both the **libc.a** routines and small or middle model compiling are required, the XENIX 3.0 shared data system calls must be used.

Small data 386 processes must specify *shmaddr* equal to zero (i.e. you must allow the system to attach the shared memory segment at whatever address it chooses).

Name

shutdn – Flushes block I/O and halts the CPU.

Syntax

```
#include <sys/filsys.h>
#include <sys/param.h>
#include <sys/types.h>

void shutdn (sblk, ntsblk, arg);
struct filsys *sblk, *nsblk;
int arg;
```

Description

shutdn 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. The super-blocks of all writable file systems are flagged 'clean', so that they can be remounted without cleaning when XENIX is rebooted. *shutdn* then prints "Normal System Shutdown" on the console and halts the CPU.

The system then stays down or reboots dependant on whether *arg* is 0 or 1.

If *sblk* is greater than 1, it specifies the address of a super-block to be written to the root device as the last I/O before the halt, provided that *nsblk* is given as its bit-wise inverse. This facility is provided to allow file system repair programs to supercede the system's copy of the root super-block with one of their own.

If *sblk* is 1, the second argument is a command and the third argument is the argument to the command. The CONFPANIC command, a system configurable system call, is given the argument 0 to stay down, or 1 to reboot. When *shutdn* is called in this way, the purpose is not to bring down the system, but rather, to give instructions to the kernel regarding the way to deal with the next panic.

shutdn locks out all other processes while it is doing its work. However, it is recommended that user processes be killed off (see *kill*(S)) before calling *shutdn* as some types of disk activity could cause file systems to not be flagged "clean".

The caller must be the super-user.

See Also

fsck(C), haltsys(C), shutdown(C), mount(S), kill(S)

Notes

This routine must be linked using the linker option **-lx.**

Name

signal – Specifies what to do upon receipt of a signal.

Syntax

```
#include <signal.h>

int (*signal (sig, func))()
int sig;
int (*func)();
```

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:

SIGHUP	01	Hangup
SIGINT	02	Interrupt
SIGQUIT	03*	Quit
SIGILL	04*	Illegal instruction (not reset when caught)
SIGTRAP	05*	Trace trap (not reset when caught)
SIGIOT	06*	I/O trap instruction
SIGEMT	07*	Emulator trap instruction
SIGFPE	08*	Floating-point exception
SIGKILL	09	Kill (cannot be caught or ignored)
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)

See number 7 below for the significance of the asterisk in the above list.

func is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are described below.

The SIG_DFL value causes termination of the process upon receipt of a signal. Upon receipt of the signal *sig*, the receiving process is to be terminated with the following consequences:

1. All of the receiving process' open file descriptors will be closed.

2. If the parent process of the receiving process is executing a *wait*, it will be notified of the termination of the receiving process and the terminating signal's number will be made available to the parent process; see *wait*(S).

3. If the parent process of the receiving process is not executing a *wait*, the receiving process will be transformed into a zombie process (see *exit*(S) for definition of zombie process).

4. The parent process ID of each of the receiving process' existing child processes and zombie processes will be set to 1. This means the initialization process (see *intro*(S)) inherits each of these processes.

5. An accounting record will be written on the accounting file if the system's accounting routine is enabled; see *acct*(S).

6. If the receiving process' process ID, tty group ID, and process group ID are equal, the signal **SIGHUP** will be sent to all of the processes that have a process group ID equal to the process group ID of the receiving process.

7. 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 user 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 a mode of 0666 modified by the file creation mask (see *umask*(S)), 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

The **SIG_IGN** value causes the process to ignore a signal. The signal *sig* is to be ignored. Note that the signal **SIGKILL** cannot be ignored.

A *function address* value causes the process to catch a 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 only argument to the signal-catching function. There are the following consequences:

1. Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted and the value of *func* for the caught signal will be set to

SIG_DFL unless the signal is **SIGILL, SIGTRAP, SIGCLD,** or **SIGPWR.**

2. When a signal 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, 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 will return a −1 to the calling process with *errno* set to EINTR.

3. Note that the signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

signal will fail if one or more of the following are true:

sig is an illegal signal number, including SIGKILL. [EINVAL]

func points to an illegal address. [EFAULT]

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.

See Also

kill(C), kill(S), pause(S), ptrace(S), wait(S), setjmp(S).

Warning

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 (not reset when caught)
SIGPWR	19	Power fail (not reset when caught)

There is no guarantee that, in future releases of XENIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of XENIX. 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' child processes will not create zombie processes when they terminate; see *exit*(S).

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** except, that while the process is executing the signal-catching function any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The **SIGCLD** affects two other system calls (*wait*(S), and *exit*(S)) 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' child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process' 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 proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

Notes

The defined constant NSIG in **signal.h** standing for the number of signals is always at least one greater than the actual number.

The calling process must make another call to *signal* after a signal is caught before another signal can be caught. If this is not done, subsequent signals are processed in the default manner (see the description for *SIG_DFL*).

Name

sigsem – Signals a process waiting on a semaphore.

Syntax

int sigsem(sem_num);
int sem_num;

Description

sigsem signals a process that is waiting on the semaphore *sem_num* that it may proceed and use the resource governed by the semaphore. *sigsem* is used in conjunction with *waitsem*(S) to allow synchronization of processes wishing to access a resource. One or more processes may *waitsem* on the given semaphore and will be put to sleep until the process which currently has access to the resource issues a *sigsem* call. If there are any waiting processes, *sigsem* causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

See Also

creatsem(S), opensem(S), waitsem(S)

System Compatibility

sigsem can only be used to signal semaphores created under XENIX Version 3.0, not for XENIX System V semaphores.

Diagnostics

sigsem returns the value (int) -1 if an error occurs. If *sem_num* does not refer to a semaphore type file, *errno* is set to ENOTNAM. If *sem_num* has not been previously opened hy *opensem*, *errno* is set to EBADF. If the process issuing a sigsem call is not the current "owner" of the semaphore (i.e., if the process has not issued a waitsem call before the sigsem), *errno* is set to ENAVAIL.

Notes

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This function must be linked using the linker option **-lx**.

Name

sinh, cosh, tanh – Performs hyperbolic functions.

Syntax

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;

Description

These functions compute the designated hyperbolic functions for real arguments.

Diagnostics

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 can be changed with the *matherr*(S) function.

See Also

matherr(S)

Notes

These routines must be linked by using the **–lm** linker option.

Name

sleep – Suspends execution for an interval.

Syntax

unsigned sleep (seconds)
unsigned 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 because scheduled wakeups occur at fixed 1-second intervals, and 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, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but 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 gone off without the intervening *sleep*.

See Also

alarm(S), nap(S), pause(S), signal(S)

Name

sputl, sgetl – Accesses long integer data in a machine-independent fashion.

Syntax

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 for all machines.

Starting at the address pointed to by *buffer*, *sgetl* retrieves the four bytes in memory and returns the long integer value in the byte ordering of the host machine.

sputl and *sgetl* provide a machine-independent way to store long numeric data in binary form in a file without converting to characters.

Name

ssignal, gsignal – Implements software signals.

Syntax

#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*(S). This facility is used by the standard C library to enable the user to indicate the disposition of error conditions, and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. An *action* for a software signal is *established* by a call to *ssignal*, and a software signal 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.

Notes

There are some additional signals with numbers outside the range 1 through 15 that 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

stat, fstat – Gets file status.

Syntax

```
#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 pathname naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the pathname 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:

```
ushort  st_mode;    /* File mode; see mknod(S) */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing */
                    /* a directory entry for this file */
dev_t   st_rdev;    /* ID of device */
                    /* This entry is defined only for */
                    /* special files */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of the file's owner */
ushort  st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time of last data modification */
time_t  st_ctime;   /* Time of last file status change */
                    /* Times measured in seconds since */
                    /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(S), *mknod*(S), *pipe*(S), *utime*(S), and *read*(S).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(S), *mknod*(S), *pipe*(S), *utime*(S), and *write*(S).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(S), *chown*(S), *creat*(S), *link*(S), *mknod*(S), *pipe*(S), *utime*(S), and *write*(S).

st_rdev Device indentification. In the case of block and character special files this contains the device major and minor numbers; in the case of shared memory and semaphores, it contains the type code. The file **/usr/include/sys/types.h** contains the macros *major*() and *minor*() for extracting major and minor numbers from *st_rdev*. See **/usr/include/sys/stat.h** for the semaphore and shared memory type code values S_INSEM and S_INSHD.

stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

buf or *path* points to an invalid address. [EFAULT]

fstat will fail if one or more of the following are true:

fildes is not a valid open file descriptor. [EBADF]

buf points to an invalid address. [EFAULT]

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

chmod(S), chown(S), creat(S), link(S), mknod(S), time(S), unlink(S)

Name

stdio – Performs standard buffered input and output.

Syntax

#include <stdio.h>
FILE *stdin, *stdout, *stderr;

Description

The *stdio* library contains an efficient, user-level I/O buffering scheme. The in-line macros *getc*(S) and *putc*(S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all 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*(S) 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 "include" file and associated with the standard open files:

stdin	Standard input file
stdout	Standard output file
stderr	Standard error file

A constant "pointer" NULL designates the null stream.

An integer constant EOF is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

Most of the functions and constants mentioned in this section of the manual are declared in that "include" file and are described elsewhere. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, and *fileno*.

See Also

open(S), close(S), read(S), write(S), ctermid(S), cuserid(S), fclose(S), ferror(S), fopen(S), fread(S), fseek(S), getc(S), gets(S), popen(S), printf(S), putc(S), puts(S), scanf(S), setbuf(S), system(S), tmpnam(S)

Diagnostics

Invalid stream pointers can cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

Name

ftok – Standard interprocess communication package.

Syntax

#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*(S), *semget*(S), and *shmget*(S) 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 refer to a project so that keys do not conflict across a given system.

ftok returns a key based on *path* and an *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 witb tbe same *id* and that it will return different keys when called with the same file name but with different *ids*.

See Also

intro(S), msgget(S), semget(S), sbmget(S)

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

stime – Sets the time.

Syntax

#include <sys/types.h>
#include <sys/timeb.h>

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.

stime will fail if the effective user ID of the calling process is not super-user. [EPERM]

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

time(S)

Name

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strdup – Performs string operations.

Syntax

```
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;

char *strdup (s)
char *s;
```

Description

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat appends a copy of string *s2* to the end of string *s1*. *strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at no more than *n* characters.

strcpy copies string *s2* to *s1*, stopping after the null character has been moved. *strncpy* copies exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

strlen returns the number of non-null characters in *s*.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or NULL 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 NULL 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. Subsequent calls with zero for the first argument, will work through the string *s1* in this way until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL is returned.

strdup returns a pointer to a duplicate copy of the string pointed to by *s*. The duplicate string is automatically allocated storage using a *malloc*(S) system call. This call allocates the exact number of bytes needed to store the string and its terminating null character.

Notes

For user convenience, all the *string* functions are declared in the **<string.h>** header file.

strcmp uses native character comparison, which is signed on some machines, unsigned on others. Thus, when one of the characters has its high-order bit set, the sign of the value returned is implementation-dependent.

All string movement is performed character by character starting at the left. Thus overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

Name

strtod, atof – Converts a string to a double-precision number.

Syntax

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

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*(S)), 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 **)0, 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*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)0)*.

See Also

ctype(S), scanf(S), strtol(S)

Diagnostics

If the correct value would cause overflow, ±**HUGE** is returned (according to the sign of the value), and *errno* is set to **ERANGE**.

If the correct value would cause underflow, zero is returned and *errno* is set to **ERANGE**.

Name

strtol, atol, atoi – Converts string to integer.

Syntax

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*. This routine scans the string up to the first character inconsistent with the base. It ignores leading white space characters as defined by *isspace* (see *ctype*(S)).

If the value of *ptr* is not (*char* **)0, *strtol* returns a pointer to the character terminating the scan at the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and *strtol* returns zero.

base is used as the base for conversion if it is positive and not greater than 36. If *base* is 16, leading zeros are ignored after an optional leading sign, and "0x" or "0X" is ignored. If *base* is zero, the string determines the base in the following manner: a leading zero indicates octal conversion after an optional leading sign; a leading "0x" or "0X" indicates hexadecimal conversion; in other cases, decimal conversion is used.

Truncation from long to int can take place upon assignment or by explicit cast.

atol(str) is equivalent to *strtol(str, (char**)0, 10)*.

atoi(str) is equivalent to *(int) strtol(str, (char**)0, 10)*.

See Also

ctype(S), scanf(S), strtod(S)

Notes

Overflow conditions are ignored.

Name

swab – Swaps bytes.

Syntax

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

Description

swab copies *nbytes* pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for transporting binary data between machines that differ in the ordering of bytes. *nbytes* should be even.

Name

swapadd – Specifies additional devices for paging and swapping.

Description

This command is available only in XENIX-386. If you have XENIX-386, see your *Release Notes* for the complete version of this reference page.

Name

sync – Updates the super-block.

Syntax

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.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

See Also

sync(C)

Name

system – Executes a shell command.

Syntax

#include <stdio.h>

int system (string)
char *string;

Description

system causes the *string* to be given to *sh*(C) 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.

Return Value

Errors, such as syntax errors, cause a non-zero return value and execution of the command file is abandoned. Otherwise, the exit status of the last command executed is returned.

See Also

sh(C), exec(S)

Diagnostics

system stops if it can't execute *sh*(C).

Name

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – Performs terminal functions.

Syntax

```
char PC;
char *BC;
char *UP;
short ospeed;

int tgetent(bp, name)
char *bp, *name;

int tgetnum(id)
char *id;

int tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;
int destcol, destline;

void tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

Description

These functions extract and use capabilities from the terminal capability data base *termcap*(F). These are low level routines; see *curses*(S) for a higher level package.

tgetent extracts the entry for terminal *name* into the buffer at *bp*. *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *tgetent* returns -1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It looks in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a pathname rather than **/etc/termcap.** This can speed up entry into programs that call *tgetent,* as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file **/etc/termcap.**

tgetnum gets the numeric value of capability *id,* returning −1 if it is not given for the terminal. *tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *tgetstr* gets the string value of capability *id,* placing it in the buffer at *area,* advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(F), except for cursor addressing and padding information.

tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline.* It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n, Ctrl-D** or **NULL** in the returned string. Programs which call *tgoto* should be sure to turn off the TAB3 bit (see *tty*(M)), since *tgoto* may now output a tab. Note that programs using termcap should turn off TAB3 anyway since some terminals use **Ctrl-I** for other functions, such as nondestructive space.) If a *%* sequence is given which is not understood, then *tgoto* returns OOPS.

tputs decodes the leading 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 external variable *ospeed* should contain the output speed of the terminal as encoded by *stty (S).* The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a NULL is inappropriate.

Files

/usr/lib/libtermcap.a	−ltermcap library
/etc/termcap	data base

See Also

curses(S), termcap(M), tty(M)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

These routines can be linked by using the **-ltermcap** linker option.

Name

terminfo – terminal description database.

Syntax

#include <curses.h>
#include <term.h>

cc -DM_TERMINFO [-DMINICURSES] ... -ltinfo [-lx]

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. Mini-Curses 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(F), terminfo(M)

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, fd)	set up new terminal of given type to output on fd
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*(M). 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 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 UNIX 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*(S). Their parameters are the same as used in *termcap*. They are emulated using the *terminfo*(M) database. They may be removed 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)

Name

time, ftime – Gets time and date.

Syntax

long time ((long *) 0)

long time (tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>

void ftime(tp)
struct timeb *tp;

Description

time returns the current system time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is nonzero, the return value is also stored in the location to which *tloc* points.

ftime returns the time in a structure (see below under *Return Value*.)

time will fail if *tloc* points to an illegal address. [EFAULT] Likewise, *ftime* will fail if *tp* points to an illegal address. [EFAULT]

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.

The *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 {
        long  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
};
```

Note that the timezone value is a system default timezone and not the value of the TZ environment variable.

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, 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.

See Also

date(C), stime(S), ctime(S)

Notes

Since *ftime* does not return the correct timezone value, its use is not recommended. See *ctime*(S) for accurate use of the TZ variable. This routine must be linked using the linker option **-lx**.

Name

times – Gets process and child process times.

Syntax

```
#include <sys/types.h>
#include <sys/times.h>

long times (tp)
struct tms *tp;
```

Description

times fills the structure pointed to by *tp* with time-accounting information. This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*(S).

All times are in clock ticks where a tick is some fraction of a second defined in *machine* (M).

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 *utimes* and *cutimes* of the child processes.

tms_cstime is the sum of the *stimes* and *cstimes* of the child processes.

times will fail if *tp* points to an illegal address. [EFAULT]

Return Value

Upon successful completion, *times* returns the elapsed real time, in clock ticks, since an arbitrary point in the past, such as the 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

exec(S), fork(S), time(S), wait(S), machine(M)

Name

tmpfile – Creates a temporary file.

Syntax

#include <stdio.h>

FILE *tmpfile ()

Description

tmpfile creates a temporary file and returns a corresponding FILE pointer. Arrangements are made so that the file will automatically be deleted when the process using it terminates. The file is opened for update.

Return Value

If the file cannot be opened, an error message is printed and a NULL pointer is returned.

See Also

creat(S), unlink(S), fopen(S), mktemp(S), tmpnam(S)

Name

tmpnam, tempnam – Creates a name for a temporary file.

Syntax

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

Description

These functions generate filenames that can safely be used for a temporary file.

tmpnam always generates a filename 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 filename.

tempnam uses *malloc*(S) to get space for the constructed filename, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free*(S) (see *malloc*(S)). If *tempnam* cannot return the expected result for any reason, i.e., *malloc*(S) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

See Also

creat(S), fopen(S), malloc(S), mktemp(S), tmpfile(S), unlink(S)

Notes

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(S) or *creat*(S) 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*(S) to remove the file when its use is ended.

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a filename 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*(S), and the filenames are chosen to make duplication by other means unlikely.

Name

sin, cos, tan, asin, acos, atan, atan2 – Performs trigonometric functions.

Syntax

```
#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 x, y;
```

Description

sin, *cos* and *tan* return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

acos returns the arc cosine in the range 0 to π.

atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

atan2 returns the arc tangent of y/x in the range $-\pi$ to π.

See Also

matherr(S)

Diagnostics

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 a TLOSS error is displayed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no error message is displayed. 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 a DOMAIN error is displayed on the standard error output.

These error-handling procedures may be changed with the *matherr*(S) function.

Notes

These routines must be linked with the **−lm** linker option.

Name

tsearch, tfind, tdelete, twalk – Manages binary search trees.

Syntax

```
#include <search.h>

char *tsearch (key, rootp, compar)
char *key;
char **rootp;
int (*compar) ( );

char *tfind (key, rootp, compar)
char *key;
char **rootp;
int (*compar)( );

char *tdelete (key, rootp, compar)
char *key;
char **rootp;
int (*compar)( );

char *twalk (root, action)
char *root;
void *action( );
```

Description

The routines *tsearch*, *tfind*, *tdelete*, and *twalk* manipulate binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to each of the elements being compared. An integer is returned less than, equal to, or greater than 0, corresponding to whether the first argument is considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so other data may be contained in the elements in addition to the compared values.

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 the value pointed to by *key* (**key*), a pointer to this datum is returned. Otherwise, **key* is inserted, and a pointer to it returned. The calling routine must store data, since only pointers are copied. *rootp* points to a variable that points to the root of the tree. A NULL value for this variable means an empty tree; in this case, this variable will be set to point to the datum at the root of the new tree.

tfind will search for a datum in the tree, returning a pointer to it if found; however, if the datum 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* is 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. *action* is called with three arguments:

- the address of the node being visited.

- a value from an enumeration data type *typedef enum { preorder, post- order, endoder, leaf} VISIT;* depending on whether this is the first, second, or third time that the node has been visited, or whether the node is a leaf. (This data type is defined in the **<search.h>** header file.)

- the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the binary search tree should be of type pointer-to-element, and cast to type pointer-to-character. The value returned should also be cast into type pointer-to-element, although it is declared as type pointer-to-character.

Examples

The following code fragment 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 length 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 root*/

main ( )
```

```
{
    char *strptr = string_space;
    struct node *nodeptr = nodes;
    void print_node ( ), twalk( );
    init 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(S), hsearch(S), lsearch(S)

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.

Warning

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 visiting a node before any of its children, after its left child and before its right, and after both children. The other nomenclatures uses preorder, inorder, and postorder to refer to the same visits.

Notes

If the calling function alters the pointer to the root, results can not be predicted.

Name

ttyname, isatty – Finds the name of a terminal.

Syntax

char *ttyname (fildes)

int isatty (fildes)
int fildes;

Description

ttyname returns a pointer to the null-terminated pathname of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

Files

/dev/*

Diagnostics

ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory **/dev**.

Notes

The return value points to static data whose content is overwritten by each call.

Name

ttyslot – Finds the slot in the utmp file of the current user.

Syntax

int ttyslot ( )

Description

ttyslot returns the index of the current user's entry in the **/etc/utmp** file.

Files

/etc/utmp

See Also

getut(S), ttyname(S)

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

uadmin – Administrative control.

Syntax

#include <sys/uadmin.h>

int uadmin (cmd, fcn, mdep)
int cmd, fcn;
char *mdep;

Description

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use.

The commands available as specified by *cmd* are:

A_SHUTDOWN
: The system is shut down. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system is shut down is specified by *fcn*. If *mdep* is non-null, then it points to a superblock to be written to the disk.

  Values of *fcn* for this *cmd* are:

AD_HALT	Halt the processor.
AD_BOOT	Reboot the system.
AD_IBOOT	Interactive reboot, prompt for system name.

A_REBOOT
: The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT
: The buffer cache is invalidated and the superblock is read in again. This should only be used during the startup process.

A_SETCONFIG
: Some internal systemwide kernel state as specified by *fcn* is set to a value as specified by *mdep*.

Values of *fnc* for this *cmd* are:

AD_BOOTPANIC	If *mdep* is 1, system panics cause the system to reboot. If *mdep* is 0, the system waits for a keystroke.

Diagnostics

Upon successful completion, the value returned depends on *cmd* as follows:

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

uadmin fails if the effective user ID is not super-user [EPERM].

Notes

AD_BOOT and AD_IBOOT do the same thing.

Name

ulimit – Gets and sets user limits.

Syntax

#include <sys/ulimit.h>

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

Description

This function provides for control over process limits. The *cmd* values available are:

UL_GFILLIM (1)
: Gets the process' file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

UL_SFILLIM (2)
: Sets the process' file size limit 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. If a process with an effective user ID other than super-user attempts to increase its file size limit, *ulimit* will fail and the limit will be unchanged. [EPERM]

UL_GMEMLIM
: Gets the maximum possible break value. If the process is a large model 80286 program, then the largest possible data size (in bytes) is returned. See *sbrk*(S).

UL_GTXTOFF
: Gets the number of bytes between the beginning of user text and the text address given by *newlimit*. In this case, *newlimit* must have type

int (*newlimit)();

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error. EINVAL indicates an invalid *cmd* value.

See Also

login(M), machine(HW), chsize(S), sbrk(S), write(S).

Notes

The file limit is only enforced on writes to regular files. Tapes, disks, and other devices of any size can be written.

The file **/etc/default/login** contains the value of **ULIMIT** set at login time by the login program. The super-user can set the maximum (increase or decrease) file size using this variable. The value is in 512 byte blocks. The default value is 4096 blocks (2 megabytes). Use even values for filesystems with 1024 byte blocks (see **machine**(HW)).

Name

umask – Sets and gets file creation mask.

Syntax

int umask (cmask)
int cmask;

Description

umask sets the process' 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.

Return Value

The previous value of the file mode creation mask is returned.

See Also

mkdir(C), mknod(C), sh(C), chmod(S), mknod(S), open(S)

Name

umount – Unmounts a file system.

Syntax

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 pathname. 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.

umount will fail if one or more of the following are true:

The process' effective user ID is not super-user. [EPERM]

spec does not exist. [ENXIO]

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

spec is not a block special device. [ENOTBLK]

spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

spec points outside the process' allocated address space. [EFAULT]

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

mount(C), mount(S)

Name

uname – Gets name of current XENIX system.

Syntax

```
#include <sys/utsname.h>

int uname (name)
struct utsname *name;
```

Description

uname stores information identifying the current XENIX system in the structure pointed to by *name*.

uname uses the structure defined in **<sys/utsname.h>**:

```
struct utsname {
        char    sysname[9];
        char    nodename[9];
        char    release[9];
        char    version[9];
        char    machine[9];
        char    reserved[15];
        unsigned short  sysorigin;
        unsigned short  sysoem;
        long    sysserial;
};
```

uname returns a null-terminated character string naming the current XENIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. Should be the same as *site name* in **/etc/systemid**. *release* and *version* further identify the operating system. *machine* identifies the processor that the system runs on, from the list: i8086, i80186, i80286, i80386, MC68000, MC68010, MC68020, NS16032, NS32032, Z8001, Z8002, VAX11780, VAX11730, PDP1123, and PDP1170. *reserved* is a reserved field. *sysorigin* and *syseom* identify the source (numbers) of the XENIX version. *sysserial* is a software serial number which may be zero if unused.

uname will fail if *name* points to an invalid address. [EFAULT]

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

See Also

uname(C)

Notes

Not all fields may be set on a particular system.

This function is a XENIX specific enhancement and may not be present on all UNIX implementations.

Name

ungetc – Pushes character back into input stream.

Syntax

#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;

Description

ungetc pushes the character *c* back on an input stream. The character will be returned by the next *getc* call on that stream. *ungetc* returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

fseek (S) erases all memory of pushed back characters.

See Also

fseek(S), getc(S), setbuf(S)

Diagnostics

ungetc returns EOF if it can't push a character back.

Name

unlink – Removes directory entry.

Syntax

int unlink (path)
char *path;

Description

unlink removes the directory entry named by the pathname pointed to by *path*.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

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. [EACCES]

The named file is a directory and the effective user ID of the process is not super-user. [EPERM]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is "." or ".." in the root directory of a mounted filesystem. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

path points outside the process' allocated address space. [EFAULT]

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.

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(C), close(S), link(S), open(S)

Name

ustat – Gets file system statistics.

Syntax

```
#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 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 */
```

ustat will fail if one or more of the following are true:

dev is not the device number of a device containing a mounted file system. [EINVAL]

buf points outside the process' allocated address space. [EFAULT]

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

stat(S), filesystem(F), fsname(M)

Notes

When using file systems from previous versions of XENIX, *fsck*(C) must be run on the file system before mounting. Otherwise the *ustat* system call will not work correctly. This only needs to be done once.

Name

utime – Sets file access and modification times.

Syntax

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

Description

path points to a pathname 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 are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct utimbuf          {
        time_t actime;   /* access time */
        time_t modtime;  /* modification time */
};
```

utime will fail if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not super-user and not the owner of the file and *times* is not NULL. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

times is not NULL and points outside the process' allocated address space. [EFAULT]

path points outside the process' allocated address space. [EFAULT]

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

stat(S)

NAME

varargs – variable argument list

Synposis

```
#include <varargs.h>

function(va_alist)
va_dcl
va_list pvar;
va_start(pvar);
f = va_arg(pvar, type);
va_end(pvar);
```

Description

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as *printf*(S)) that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to denote a variable argument list.

va_dcl is a declaration for va_alist. Note that there is no semicolon after va_dcl.

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start(*pvar*) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) 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 since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by va_start ... va_end, are possible.

Example

```
#include <stdio.h>
#include <varargs.h>

main()
{
        show(2, 3.1, "but", 4.1, "end");
        show(1, 5.9, "hello");
        show(4, 6.2, "oops", 5.3, "blah", 5.1, "lovely", 2.3, "madrigal");
}

/*
 * the first argument is an int which tells how many pairs follow.
 * the pairs are doubles and character pointers
 *
 * remember that when variables are passed to functions
 * floats are promoted to doubles and chars to ints.
 */
show(n, va_alist)
int n;
va_dcl
{
        va_list ap;
        int i;
        double f;
        char *p;

        va_start(ap);
        for (i = 0; i < n; ++i) {
                f = va_arg(ap, double);
                p = va_arg(ap, char *);
                printf("%4.1f %s\n", f, p);
        }
        va_end(ap);
}
```

Notes

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, *excel* passes a 0 to signal the end of the list. *printf* can tell how many arguments are supposed to be there by the format of the list.

Name

vprintf, vfprintf, vsprintf – Prints formatted output of a *varargs* argument list.

Syntax

```
#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 in **varargs.h**.

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(fmt, args);
  va_end(args);
  (void)abort( );
}
```

Files

/usr/include/varargs.h

See Also

printf(S)

Name

wait – Waits for a child process to stop or terminate.

Syntax

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

Description

wait suspends the calling process until it receives a signal that is to be caught (see *signal*(S)), or until any one of the calling process' child processes stops in a trace mode (see *ptrace*(S)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is nonzero, 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 and if the child process 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 be zero and the low-order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *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*(S).
>
> 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 200) is set, a "core image" will have been produced; see *signal*(S).

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; see *intro*(S).

wait will fail and return immediately if one or more of the following are true:

The calling process has no existing unwaited-for child processes. [ECHILD]

stat_loc points to an illegal address. [EFAULT]

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. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

See Also

exec(S), exit(S), fork(S), pause(S), signal(S)

Warning

See *Warning* in *signal*(S).

Name

waitsem, nbwaitsem – Awaits and checks access to a resource governed by a semaphore.

Syntax

int waitsem(sem_num);
int sem_num;

int nbwaitsem(sem_num);
int sem_num;

Description

waitsem gives the calling process access to the resouree governed by the semaphore *sem_num*. If the resource is in use by another process, *waitsem* will put the process to sleep until the resource becomes available; *nbwaitsem* will return the error ENAVAIL. *waitsem* and *nbwaitsem* are used in conjunction with *sigsem* to allow synchronization of processes wishing to access a resource. One or more processes may *waitsem* on the given semaphore and will be put to sleep until the process which currently has access to the resource issues *sigsem*. *sigsem* causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

System Compatibility

waitsem can only be used to synchronize semaphores created under XENIX Version 3.0, not for XENIX System V semaphores.

See Also

creatsem(S), opensem(S), sigsem(S)

Diagnostics

waitsem returns the value (int) -1 if an error occurs. If *sem_num* has not been previously opened by a call to *opensem* or *creatsem*, *errno* is set to EBADF. If *sem_num* does not refer to a semaphore type file, *errno* is set to ENOTNAM. All processes waiting (or attempting to wait) on the semaphore return with *errno* set to ENAVAIL when the process controlling the semaphore exits without relinquishing control (thereby leaving the resource in an undeterminate state). If a process does two *waitsems* in a row without doing an intervening *sigsem*, *errno* is set to EINVAL.

Notes

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine must be linked with the linker option -lx.

Name

write – Writes to a file.

Syntax

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

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*.

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 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 is set to the end of the file prior to each write.

write will fail and the file pointer will remain unchanged if one or more of the following are true:

> *fildes* is not a valid file descriptor open for writing. [EBADF]
>
> An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]
>
> An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See *ulimit*(S). [EFBIG]
>
> *buf* points outside the process' allocated address space. [EFAULT]
>
> A signal was caught during the *write* system call. [EINTR]
>
> There is no free space remaining on the device containing the file.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(S)) 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 nonzero number of bytes gives a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, the write will fail if a write of *nbyte* bytes exceeds a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then a write to a full pipe (or FIFO) returns a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) block until space becomes available.

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.

See Also

creat(S), dup(S), lseek(S), open(S), pipe(S), ulimit(S)

Notes

Writing a region of a file locked with *locking* causes *write* to hang indefinitely until the locked region is unlocked.

Name

xlist, fxlist – Gets name list entries from files.

Syntax

```
#include <a.out.h>

int xlist(filename, xl)
char *filename;
struct xlist xl[ ];

#include <a.out.h>
#include <stdio.h>
int fxlist(fp, xl)
FILE *fp;
struct xlist xl[ ];
```

Description

fxlist performs the same function as *xlist*, except that *fxlist* accepts a pointer to a previously opened file intead of a filename.

xlist examines the name list in the given executable output file and selectively extracts a list of values. The name list structure *xl* consists of an array of *xlist* structures containing names, types, values, and segment values (if applicable). The list is terminated by either a pointer to a null name or a null pointer. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted into the next two fields. The segment value (if it exists) is inserted in the third field. If the name is not found, both entries are set to zero. See *a.out*(F) for a discussion of the xlist structure.

x.out and *a.out* formats are understood, as well as 8086 relocatable and *x.out* segmented formats.

If the symbol table is in *a.out* format, and if the symbol name given to *xlist* is longer than eight characters, only the first eight characters are used for comparison. In all other cases, the name given to *xlist* must be the same length as a name list entry in order to match.

If two or more symbols happen to match the name given to *xlist*, then the type and value used will be those of the last symbol found.

See Also

a.out(F)

Diagnostics

xlist returns -1 and sets all type entries to zero if the file cannot be read, is not an object file, or contains an invalid name list. Otherwise, *xlist* returns zero. A return value of zero does *not* indicate that any or all of the given symbols were found.

Contents

strset	Sets all characters in a string to one charater.
strupr	Converts lowercase characters to uppercase.
tell	Gets the current position of the file pointer.
ultoa	Converts numbers to characters.
ungetch	Returns a character to the console buffer.

Name

intro – Introduction to DOS cross development functions.

Description

This section contains manual pages describing functions that can be used to create program files executable under the DOS operating system. These functions are specifically for use in creating DOS executable program files.

Source files containing these functions must be compiled with the **-dos** flag. For example:

```
cc -dos test.c
```

The resulting *a.out* file is executable only under the DOS operating system. These functions cannot be used to create program files executable under XENIX.

Name

bdos – Invokes a DOS system call.

Syntax

#include <dos.h>

int bdos (dosfn, dosdx, dosal);
int dosfn;
unsigned int dosdx;
unsigned int dosal;

Description

The *bdos* function invokes the MS-DOS system call specified by *dosfn* after placing the values specified by *dosdx* and *dosal* in the DX and AL registers, respectively. *bdos* executes an INT 21H instruction to invoke the system call. When the system call returns, *bdos* returns the content of the AX register.

bdos is intended to be used to invoke DOS system calls that either take no arguments or only take arguments in the DX (DH,DL) and/or AL registers.

Return Value

bdos returns the value of the AX register after the system call has completed.

See Also

intdos (DOS), intdosx(DOS)

Example

```
#include <bdos.h>

char *buffer = "Enter file name:$";

    /* AL is not needed, so 0 is used */
bdos (9, (unsigned) buffer, 0);
```

Notes

This call should not be used to invoke system calls that indicate errors by setting the carry flag. Since C programs do not have access to this flag, the status of the return value cannot be determined. The *intdos* function should be used in these cases.

This call must be compiled with the **-dos** flag.

Name

cgets – Gets a string.

Syntax

#include <conio.h>

char *cgets (str);
char * str;

Description

The *cgets* function reads a string of characters directly from the console and stores the string and its length in the location pointed to by *str*. The *str* must be a pointer to a character array. The first element of the array, *str*[0], must contain the maximum length (in characters) of the string to be read. The array must have enough elements to hold the string, a terminating null character ('\0'), and two additional bytes.

cgets continues to read characters until a carriage return/linefeed combination (CR-LF) is read, or the specified number of characters have been read. The string is stored starting at *str*[2]. If a CR-LF combination is read, it is replaced with a null chracter ('\0') before being stored. *cgets* then stores the actual length of the string in the second array element, *str*[1].

Return Value

cgets returns a pointer to the start of the string, which is at *str*[2]. There is no error returned.

See Also

getch(DOS), getche(DOS)

Example

```
#include <conio.h>

char buffer[82];
char *result;
int numread;
   .
   .
   .
*buffer = 80;  /* maximum number of chracters */
               /* note that *buffer is equivalent
               ** to buffer[0]
               */
/* The following statements input a string from the
** keyboard and find its length.
*/

result = cgets(buffer);
numread = buffer[1];

/* Result points to the string, and numread is its
** length (not counting the carriage return, which has
** been replaced by a null chracter).
*/
```

Notes

This call must be compiled with the **-dos** flag.

Name

cprintf – Formats output.

Syntax

#include <conio.h>

int cprintf (format[*arg*...]);
char *format;

Description

The *cprintf* function formats and prints a series of characters and values directly to the console, using the *putch* function to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format*. The *format* has the same form and function as the *format* argument for the *printf* function; see the *printf* reference page for a description of the *format* and arguments.

Return Value

cprintf returns the number of characters printed.

See Also

fprintf(S), printf(S), sprintf(S)

Example

```
#include <conio.h>

int i = -16, j = 29;
unsigned int k = 511;

/* The following statement prints i=-16, j=0x1d, k=511 */

cprintf ("i=%d, j=%#x, k=%u\n",i,j,k);
```

Notes

Unlike the *fprintf*, *printf*, and *sprintf* functions, *cprintf* does not translate linefeed (LF) characters into carriage return/linefeed combinations (CR-LF) on output.

This call must be compiled with the **-dos** flag.

Name

cputs – Puts a string to the console.

Syntax

#include <conio.h>

void cputs (str);
char *str;

Description

The *cputs* function writes the null-terminated string pointed to by *str* directly to the console. Note that a carriage return/linefeed combination (CR-LF) is not automatically appended to the string after writing.

Return Value

There is no return value.

See Also

putch(DOS)

Example

```
#include <conio.h>

char *buffer = "Insert data disk in drive a: \r\n";

/* The following statement outputs a prompt to the
** console.
*/

cputs (buffer);
```

Notes

This call must be compiled with the **-dos** flag.

Name

cscanf – Converts and formats console input.

Syntax

#include <conio.h>

int cscanf (format[*arg...* **]);**
char *format;

Description

The *cscanf* function reads data directly from the console into the locations given by the *arguments* (if any), using the *getche* function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format*. The *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for the *scanf* function.

Return Value

cscanf returns the number of fields that were successfully converted and assigned. The return value does not include fields which were read but not assigned.

The return value is EOF for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

See Also

fscanf(S), scanf(S), sscanf(S)

Example

```
#include <conio.h>

int result;
char buffer[20];
  .
  .
  .
cprintf ("Please enter file name: ");

/* The following statement stores string input
** from the keyboard.
*/

result = cscanf ("%19s",buffer);

/* Result is the number of correctly matched input
** fields. It is zero if none could be matched.
*/
```

Notes

This call must be compiled with the **-dos** flag.

Name

dosexterr – Gets DOS error messages

Summary

#include <dos.h>

int dosexterr (buffer);
struct DOSERROR *buffer;

Description

The *dosexterr* function obtains the register values returned by the MS-DOS system call 59H and stores the values in the structure pointed to by *buffer*. This function is useful when making system calls under MS-DOS Version 3.0 or later, which offers extended error handling. See your MS-DOS reference for details on MS-DOS system calls.

The structure type DOSERROR is defined in **dos.h** as follows:

```
struct DOSERROR {
        int exterror;
        char class;
        char action;
        char locus;
        };
```

Giving a NULL pointer argument causes *dosexterr* to return the value in AX without filling in the structure fields.

Return Value

The *dosexterr* function returns the value in the AX register (identical to the value in the *exterror* structure field).

See Also

perror(S)

Example

```
#include <dos.h>
#include <fcntl.h>
#include <stdio.h>

struct DOSERROR doserror;
int fd;

if ((fd = open ("test.dat", O_RDONLY)) == -1) {
    dosexterr (&doserror);
    printf ("error=%d, class=%d, action=%d, locus=%d\n",
        doserror.exterror, doserror.class,
        doserror.action, doserror.locus);
    }
```

Notes

The *dosexterr* function should only be used under MS-DOS Version 3.0 or later.

This call must be compiled with the **-dos** flag.

Name

eof – Determines end-of-file.

Syntax

#include <io.h>

int eof (handle);
int handle;

Description

The *eof* function determines whether end-of-file has been reached for the file associated with *handle.*

Return Value

eof returns the value 1 if the current position is end-of-file, 0 if it is not. A return value of −1 indicates an error; in this case *errno* is set to EBADF, indicating an invalid file handle.

See Also

ferror(S), perror(S)

Example

```
#include <io.h>
#include <fcntl.h>

int fh, count;
char buf[10];

fh = open ("data",ORDONLY);
    .
    .
    .

/* The following statement tests for an end-of-file condition
** before reading.
*/

while (!eof (fh)) {
    count = read (fh, buf, 10);
    .
    .
    .
    }
```

Notes

This call must be compiled with the **-dos** flag.

Name

exit – Terminates the calling process.

Syntax

#include <process.h>

void exit (status);

void _exit (status);

int status;

Description

The *exit* and *_exit* functions terminate the calling process. *exit* flushes all buffers and closes all open files before terminating the process. *_exit* terminates the process without flushing stream buffers. *Status* is typically given the value 0 to indicate a normal exit and set to some other value to indicate an error.

Although the *exit* and *_exit* calls do not return a value, the low-order byte of *status* is made available to the waiting parent process, if there is one, after the calling process exits. If there is no parent process waiting on the exiting process, the *status* value is lost.

Return Value

There is no return value.

See Also

abort(S), exec(S), spawn(DOS)

Example

```
#include <process.h>
#include <stdio.h>

FILE *stream;
      .
      .
      .
/* The following statements cause the process to
** terminate, after flushing buffers and closing
** open files, if another file cannot be opened.
*/

if ((stream = fopen ("data","r")) == NULL) {
    perror ("couldn't open data file");
    exit (1);
    }

/* The following statements cause the process to
** terminate immediately if a file cannot be opened.
*/

if ((stream = fopen ("data","r")) == NULL) {
    perror ("couldn't open data file");
    exit (1);
    }
```

Notes

These calls must be compiled with the **-dos** flag.

Name

fclose, fcloseall – Closes streams.

Syntax

#include <stdio.h>

int fclose (stream);
FILE *stream;

int fcloseall ( );

Description

The *fclose* and *fcloseall* functions close a stream or streams. All buffers associated with the stream(s) are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned using *setbuf* are not automatically released.

The *fclose* function closes the given *stream*. The *fcloseall* function closes all open streams except *stdin, stdout, stderr, stdaux,* and *stdprn*.

Return Value

fclose returns 0 if the stream is successfully closed. *fcloseall* returns the total number of streams closed. Both functions return EOF to indicate an error.

See Also

close(S), fopen(S), fclose(S)

Example

```
#include <stdio.h>

FILE *stream;
int numclosed;

stream = fopen ("data","r");
   .
   .
   .

/* The following statement closes the stream.
*/

fclose (stream);


/* The following statement closes all streams except
** stdin, stdout, stderr, stdaux, and stdprn.
*/

numclosed = fcloseall ( );
```

Notes

These calls must be compiled with the **-dos** flag.

Name

fgetc, fgetchar – Gets a character from a stream.

Syntax

#include <stdio.h>

int fgetc (stream);
FILE *stream;

int fgetchar ( );

Description

The *fgetc* function reads a single character from the input *stream* at the current position and increments the associated file pointer (if any) to point to the next character. *fgetchar* is equivalent to *fgetc*(*stdin*).

Return Value

fgetc and *fgetchar* return the character read. A return value of EOF may indicate an error or end-of-file; however, the EOF value is also a legitimate integer value, so *feof* or *ferror* should be used to verify an error or end-of-file condition.

See Also

putc(S), fputchar(DOS), getc(S)

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;
   .
   .
   .
/* The following statements gather a line of input from
** a stream.
*/

for (i = 0; (i < 80) && ((ch = fgetc (stream)) != EOF) &&
    (ch != '\n'); i++)
    buffer[i] = ch;

buffer[i] = '\0';

/* "fgetchar ( )" could be used instead of "fgetc (stream)" in
** the for statement above to gather a line of input from
** stdin (equivalent to "fgetc (stdin)").
*/
```

Notes

fgetc and *fgetchar* are identical to *getc* and *getchar*, but are functions, not macros.

These calls must be compiled with the **-dos** flag.

Name

filelength – Gets the length of a file.

Syntax

#include <io.h>

long filelength (handle);
int handle;

Description

The *filelength* function returns the length in bytes of the file associated with the given *handle.*

Return Value

filelength returns the file length in bytes. A return value of -1L indicates an error, and *errno* is set to EBADF to indicate an invalid file handle.

See Also

chsize(S), ferror(S), stat(S)

Example

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
long length;

stream = fopen ("data","r");
  .
  .
  .
/* The following statements attempt to determine the
** length of a file associated with a stream.
*/

length = filelength (fileno (stream));

if (length == -1L)
    perror ("filelength failed");
```

Notes

This call must be compiled with the **-dos** flag.

Name

flushall – Flushes all output buffers.

Syntax

#include <stdio.h>

int flushall ( );

Description

The function *flushall* causes the contents of all buffers associated with open output streams to be written to the associated files. All streams remain open after the call.

Return Value

flushall returns the number of open streams (input and output). There is no error return.

See Also

fclose(S)

Example

```
#include <stdio.h>

int numflushed;
  .
  .
  .

/* The following statement resolves any pending i/o on
** all streams.
*/

numflushed = flushall ( );
```

Notes

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

This call must be compiled with the **-dos** flag.

Name

fp_off, fp_seg – Return offset and segment.

Syntax

#include <dos.h>

unsigned FP_OFF(longptr);

unsigned FP_SEG(longptr);

char far *longptr;

Description

The *FP_OFF* and *FP_SEG* macros return the offset and segment, respectively, of the long pointer *longptr*.

Return Value

FP_OFF returns an unsigned integer value representing an offset. *FP_SEG* returns an unsigned integer value representing a segment address.

See Also

segread(DOS)

Example

```
#include <dos.h>

char far *p;
unsigned int sp;
unsigned int op;
        .
        .
        .
sp = FP_SEG(p);
op = FP_OFF(p);
```

Notes

These calls must be compiled with the **-dos** flag.

Name

fputc, fputchar – Write a character to a stream.

Syntax

#include <stdio.h>

int fputc (c, stream);
int c;
FILE *stream;

int fputchar (c);
int c;

Description

The *fputc* function writes the single character *c* to the output *stream* at the current position. *fputchar* is equivalent to *fputc*(*c, stdout*).

Return Value

fputc and *fputchar* return the character written. A return value of EOF may indicate an error. However, since the EOF value is also a legitimate integer value, use *ferror* to verify an error condition.

See Also

fgetc(DOS), getc(S), putc(S)

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;
  .
  .
  .

/* The following statements write the contents of a buffer to
** a stream. Note that the output occurs as a side effect
** within the for statement's second expression, so the
** statement body is null.
*/

for (i = 0; (i < 81) &&
    ((ch = fputc (buffer[i],stream)) != EOF); i++)
    ;

/* "fputchar ( )" could be used instead of "fputc (stream)"
** in the for statement above to write the buffer to stdout
** (equivalent to "fputc (stdout)").
*/
```

Notes

fputc and *fputchar* are identical to *putc* and *putchar*, but are functions, not macros.

These calls must be compiled with the **-dos** flag.

Name

getch – Gets a character.

Syntax

#include <conio.h>

int getch ( );

Description

The *getch* function reads, without echoing, a single character directly from the console. Characters typed are not echoed. If a CONTROL-C is typed, the system executes an INT 23H (CONTROL-C exit).

Return Value

getch returns the character read. There is no error return.

See Also

cgets (DOS), getche (DOS), getchar(S)

Example

```
#include <conio.h>
#include <ctype.h>

int ch;

/* This loop gets characters from the keyboard until a
** non-blank character is seen. Preceding blank
** characters are discarded.
*/

do {
    ch = getch ( );
    } while (isspace (ch));
```

Notes

This call must be compiled with the **-dos** flag.

Name

getche – Gets and echoes a character.

Syntax

#include <conio.h>

int getche ( );

Description

The *getche* function reads a single character from the console and echoes the character read. If a CONTROL-C is typed, the system executes an INT 23H (CONTROL-C exit).

Return Value

getche returns the character read. There is no error return.

See Also

cgets(DOS), getch(DOS)

Example

```
#include <conio.h>
#include <ctype.h>

int ch;

/* Get a character from the keyboard and echo it to the
** console. If it is an upper case letter, convert it
** to lower case and write over the old character.
*/

ch = getche ( );

if (isupper (ch))
    cprintf ("\b%c",tolower (ch));
```

Notes

This call must be compiled with the **-dos** flag.

Name

inp – Returns a byte.

Syntax

#include <conio.h>

int inp (port);
unsigned port;

Description

The *inp* function reads one byte from the input port specified by *port*. The *port* argument can be any unsigned integer number in the range 0 to 65,535.

Return Value

inp returns the byte read from *port*. There is no error return.

See Also

outp(DOS)

Example

```
#include <conio.h>

unsigned port;
char result;
    .
    .
    .

/* The following statement inputs a byte from the port
** that 'port' is currently set to.
*/

result = inp (port);
```

Notes

This call must be compiled with the **-dos** flag.

Name

int86 – Executes an interrupt.

Syntax

```
#include <dos.h>

int int86(intno, inregs, outregs);
int intno;
union REGS *inregs;
union REGS *outregs;
```

Description

The *int86* function executes the 8086 software interrupt specified by the interrupt number *intno.* Before executing the interrupt, *int86* copies the contents of *inregs* to the corresponding registers. After the interrupt returns, the function copies the current register values to *outregs.* It also copies the status of the system carry flag to the *cflag* field in *outregs.* The *inregs* and *outregs* arguments are unions of type *REGS*. The union type is defined in the include file **dos.h.**

Int86 is intended to be used to invoke DOS interrupts directly.

Return Value

The return value is the value in the AX register after the interrupt returns. If the *flag* field in *outregs* is nonzero, an error has occurred and the *doserrno* variable is also set to the corresponding error code.

See Also

bdos(DOS), intdos(DOS), intdosx(DOS), int86x(DOS)

Example

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

/*
 * Use int86 routine to generate a CONTROL-C interrupt
 * (interrupt number 0x23) which would be caught by the
 * interrupt handling routine inthandler. Note that the
 * values in the regs struct do not matter for this
 * interrupt.
 */

#define CNTRLC 0x23
int inthandler (int);
union REGS regs;
    .
    .
    .

signal (SIGINT, inthandler);
    .
    .
    .
int86(CNTRLC, &regs, &regs);
```

Notes

Segment registers are not included in *inregs* or *outregs*.

This call must be compiled with the **-dos** flag.

Name

int86x – Executes an interrupt.

Syntax

#include <dos.h>

int int86x (intno, inregs, outregs, segregs);
int intno;
union REGS *inregs;
union REGS *outregs;
struct SREGS *segregs;

Description

The *int86x* function executes the 8086 software interrupt specified by the interrupt number *intno*. Unlike the *int86* function, *int86x* accepts segment register values in *segregs*, letting programs that use long model data segments or far pointers specify which segment or pointer should be used during the system call.

Before executing the specified interrupt, *int86x* copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the interrupt returns, the function copies the current register values to *outregs* and restores DS. It also copies the status of the system carry flag to the *cflag* field in *outregs*. The *inregs* and *outregs* arguments are unions of type *REGS*. The *segregs* argument is a structure of type *SREGS*. These types are defined in the include file **dos.h.**

int86x is intended to be used to directly invoke DOS interrupts that take an argument in the ES register, or take a DS register value that is different than the default data segment.

Return Value

The return value is the value in the AX register after the interrupt returns. If the *flag* field in *outregs* is nonzero, an error has occurred and the *doserrno* variable is also set to the corresponding error code.

See Also

bdos(DOS), intdos(DOS), intdosx(DOS), int86(DOS), segread(DOS), FP_SEG(DOS)

Example

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

/*
 * Use int86x routine to generate an interrupt 0x21 (system
 * call), which invokes the DOS 'Change Attributes' system
 * call.  The int86x routine is used because the filename to
 * be referenced may be in a segment other than the default
 * data segment (it is referenced by a far pointer), so the
 * DS register must be explicitly set via the SREGS struct.
 */

#define SYSCALL     0x21        /* INT 21H invokes system
                                calls */
#define CHANGE_ATTR 0x43        /* system call 43H - change
                                attributes */

char far *filename;             /* filename in 'far' data
                                segment */

union REGS inregs, outregs;
struct SREGS segregs;
int result;
    .
    .
    .
inregs.h.ah = CHANGE_ATTR;      /* AH is system call
                                number */
inregs.h.al = 0;                /* AL is function (get
                                attributes) */
inregs.x.dx = FP_OFF(filename); /* DS:DX points to file
                                name */
segregs.ds = FP_SEG(filename);
result = int86x (SYSCALL, &inregs, &outregs, &segregs);
if (outregs.x.cflag) {
    printf ("can't get attributes of file; error number %d\n",
        result);
    exit (1);
    }
else {
    printf ("Attribs = %#x\n", outregs.x.cx);
    }
```

Notes

Segment values for the *segregs* argument can be obtained by using either the *segread* function or the *FP_SEG* macro.

This call must be compiled with the **-dos** flag.

Name

intdos – Invokes a DOS system call.

Syntax

#include <dos.h>

int intdos (inregs, outregs);
union REGS *inregs;
union REGS *outregs;

Description

The *intdos* function invokes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The *inregs* and *outregs* arguments are unions of type *REGS*. The union type is defined in the include file **dos.h**.

To invoke a system call, *intdos* executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* to the corresponding registers. After the INT instruction returns, *intdos* copies the current register values to *outregs*. It also copies the status of the system carry flag to the *cflag* field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

intdos is intended to be used to invoke DOS system calls that take arguments in registers other than DX (DH/DL) and AL, or to invoke system calls that indicate errors by setting the carry flag.

Return Value

intdos returns the value of the AX register after the system call has completed. If the *flag* field in *outregs* is nonzero, an error has occurred and *doserrno* is also set to the corresponding error code.

See Also

bdos(DOS), int86(DOS), int86x(DOS), intdosx(DOS)

Example

```
#include <dos.h>
#include <stdio.h>

union REGS inregs, outregs;
     .
     .
     .

/* The following statements get the current date using
** dos function call 2a hex.
*/

inregs.h.ah = 0x2a;
intdos (&inregs,&outregs);
printf ("date is %d/%d/%d\n",outregs.h.dh,outregs.h.dl,
    outregs.x.cx);
```

Notes

This call must be compiled with the **-dos** flag.

Name

intdosx – Invokes a DOS system call.

Syntax

#include <dos.h>

int intdosx (inregs, outregs, segregs);
union REGS *inregs;
union REGS *outregs;
struct SREGS *segregs;

Description

The *intdosx* function invokes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. Unlike the *intdos* function, *intdosx* accepts segment register values in *segregs*, letting programs that use long model data segments or far pointers specify which segment or pointer should be used during the system call. The *inregs* and *outregs* arguments are unions of type *REGS*. The *segregs* argument is a structure of type *SREGS*. These types are defined in the include file **dos.h**.

To invoke a system call, *intdosx* executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the INT instruction returns, *intdosx* copies the current register values to *outregs* and restores DS. It also copies the status of the system carry flag to the *cflag* field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

intdosx is intended to be used to invoke DOS system calls that take an argument in the ES register, or that take a DS register value that is different from the default data segment.

Return Value

intdosx returns the value of the AX register after the system call has completed. If the *flag* field in *outregs* is nonzero, an error has occurred and *doserrno* is also set to the corresponding error code.

See Also

bdos (DOS), intdos (DOS), segread (DOS), FP_SEG(DOS)

Example

```
#include <dos.h>

union REGS inregs, outregs;
struct SREGS segregs;
char far *dir = "/test/bin";

/* The following statements change the current working
** directory with dos function call 3b hex.
*/

inregs.h.ah = 0x3b;                 /* change directory */
inregs.x.dx = FPOFF(dir);           /* file name offset */
segregs.ds = FPSEG(dir);            /* file name segment */
intdosx (&inregs,&outregs,&segregs);
```

The above example must be compiled using the **−Me** flag.

Notes

Segment values for the *segregs* argument can be obtained by using either the *segread* function or the *FP_SEG* macro.

This call must be compiled with the **-dos** flag.

Name

isatty – Checks for a character device.

Syntax

#include <io.h>

int isatty (handle);
int handle;

Description

The *isatty* function determines whether the given *handle* is associated with a character device (that is, a terminal, console, printer or serial port).

Return Value

isatty returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

Example

```
#include <io.h>

int fh;
long loc;
    .
    .
    .
if (isatty (fh) == 0)
    loc = tell (fh);    /* if not a device, get current
                        ** position
                        */
```

Notes

This call must be compiled with the **-dos** flag.

Name

itoa – Converts integers to characters.

Syntax

#include <stdlib.h>

char *itoa (value, string, radix);
int value;
char *string;
int radix;

Description

The *itoa* function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value*. It must be in the range 2–36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–).

Return Value

itoa returns a pointer to *string*. There is no error return.

See Also

ltoa(DOS), ultoa(DOS)

Example

```
#include <stdlib.h>

int radix = 8;
char buffer[20];
char *p;
      .
      .
      .
p = itoa (-3445,buffer,radix); /* p = "171213" */
```

 ITOA (DOS)

Notes

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 17 bytes.

This call must be compiled with the **-dos** flag.

Name

kbhit – Checks the console for a keystroke.

Syntax

#include <conio.h>

int kbhit ( );

Description

The *kbhit* function checks the console for a recent keystroke.

Return Value

kbhit returns a nonzero value if a key has been pressed. Otherwise, it returns zero.

Example

```
#include <conio.h>

int result;

/* The following statement tests to see if a key has
** been hit.
*/

result = kbhit ( );

/* If result is nonzero, a keystroke is waiting in the
** buffer. It can be fetched with getch or getche.
** If getch or getche were called without first checking
** kbhit, the program might pause while waiting for
** input.
*/
```

Notes

This call must be compiled with the **-dos** flag.

Name

labs – Returns the absolute value of a long integer.

Syntax

```
#include <stdlib.h>

loug labs (n);
long n;
```

Description

The *labs* function produces the absolute value of its long integer argument *n*.

Return Value

labs returns the absolute value of its argument. There is no error return.

See Also

abs(DOS), fabs(DOS), hypot(S)

Example

```
#include <stdlib.h>

long x, y;

x = -41567L;
y = labs (x); /* y = 41567L */
```

Notes

This call must be compiled with the **-dos** flag.

Name

ltoa – Converts long integers to characters.

Syntax

```
#include <stdlib.h>

char *ltoa (value, string, radix);
long value;
char *string;
int radix;
```

Description

The *ltoa* function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value*. It must be in the range 2–36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–).

Return Value

ltoa returns a pointer to *string*. There is no error return.

See Also

itoa (DOS), ultoa (DOS)

Example

```
#include <stdlib.h>

int radix = 10;
char buffer[20];
char *p;

p = ltoa (-344115L,buffer,radix);/* p = "-344115" */
```

Notes

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

This call must be compiled with the **-dos** flag.

Name

mkdir – Creates a new directory.

Syntax

#include <direct.h>

int mkdir (pathname);
char *pathname;

Description

The *mkdir* function creates a new directory with the specified *pathname*. Only one directory can be created at a time, so only the last component of *pathname* can name a new directory.

Return Value

mkdir returns the value 0 if the new directory was created. A return value of −1 indicates an error, and *errno* is set to one of the following values:

Value	**Meaning**
EACCES	Directory not created: the given name is the name of an existing file, directory, or device.
ENOENT	Pathname not found.

See Also

chdir(S), rmdir(DOS)

Example

```
#include <direct.h>

int result;

/* The following two statements create two new directories:
** one at the root on drive b:, and one in the "tmp"
** subdirectory of the current working directory.
*/

result = mkdir ("b:/tmp");       /* "b:\\tmp" could also
  .                              ** be used
  .                              */
  .
result = mkdir ("tmp/sub");      /* "tmp\\sub" could also
                   ** be used
                   */
```

Notes

This call must be compiled with the **-dos** flag.

Name

movedata – Copies bytes from a specific address.

Syntax

#include <memory.h>

void movedata (srcseg, srcoff, destseg, destoff, nbytes);
int srcseg;
int srcoff;
int destseg;
int destoff;
unsigned nbytes;

Description

The *movedata* function copies *nbytes* bytes from the source address specified by *srcseg:srcoff* to the destination address specified by *destseg:destoff*.

movedata is intended to be used to move far data in small or medium model programs where segment addresses of data are not implicitly known. In large model programs, the *memcpy* function can be used since segment addresses are implicitly known.

Return Value

There is no error return.

See Also

memory(S), segread(DOS), FP_OFF(DOS)

Example

```
#include <memory.h>
#include <dos.h>

char far *src;
char far *dest;
    .
    .
    .
/* The following statement move 512 bytes of data from
** src to the dest.
*/

movedata(FP_SEG(src), FP_OFF(src), FP_SEG(dest),
    FP_OFF(dest, 512);

x = -14.87654321;
y = modf (x,&n);         /* y = -0.87654321, n = -14.0 */
```

Notes

Segment values for the *srcseg* and *destseg* arguments can be obtained by using either the *segread* function or the *FP_SEG* macro.

movedata does not handle all cases of overlapping moves correctly (overlapping moves occur when part of the destination is the same memory area as part of the source). Overlapping moves are handled correctly in the *memcpy* function.

This call must be compiled with the **-dos** flag.

Name

outp – Writes a byte to an output port.

Syntax

#include <conio.h>

int outp (port, value);
unsigned port;
int value;

Description

The *outp* function writes the specified *value* to the output port specified by *port*. The *port* argument can be any unsigned integer in the range 0 to 65,535. *value* can be any integer in the range 0 to 255.

Return Value

outp returns *value*. There is no error return.

See Also

inp(DOS)

Example

```
#include <conio.h>

int port, byte_val;
     .
     .
     .
/* The following statement outputs a byte to the port
** that 'port' is currently set to.
*/

outp (port,byte_val);
```

Notes

This call must be compiled with the **-dos** flag.

Name

putch – Writes a character to the console.

Syntax

#include <conio.h>

void putch (c)
int c;

Description

The *putch* function writes the character *c* directly to the console.

Return Value

There is no return value.

See Also

cprintf(DOS), getch(DOS), getche(DOS)

Example

```
#include <conio.h>

/* This example shows how the getche function could be defined
** using putch and getch.
*/

int getche ( )
{
    int ch;

    ch = getch ( );
    putch (ch);
    return (ch);
}
```

Notes

This call must be compiled with the **-dos** flag.

Name

rename – renames a file or directory.

Syntax

#include <io.h>

int rename (newname, oldname);
char *newname;
char *oldname;

Description

The *rename* function renames the file or directory specified by *old-name* to the name given by *newname*. *oldname* must specify the pathname of an existing file or directory. *Newname* must not specify the name of an existing file or directory.

The *rename* function can be used to move a file from one directory to another by giving a different pathname in the *newname* argument. However, files cannot be moved from one device to another (for example, from Drive A to Drive B). Directories can only be renamed, not moved.

Return Value

rename returns 0 if it is successful.

See Also

creat(S), fopen(DOS), open(S)

Example

```
#include <io.h>

int result;

/* The following statement changes the file "data" to
** have the name "input".
*/
result = rename ("input", "data"):
```

Notes

This call must be compiled with the **-dos** flag.

Name

rmdir – Deletes a directory.

Syntax

```
#include <direct.h>

int rmdir (pathname);
char *pathname;
```

Description

The *rmdir* function deletes the directory specified by *pathname*. The directory must be empty, and it must not be the current working directory or the root directory.

Return Value

rmdir returns the value 0 if the directory is successfully deleted. A return value of –1 indicates an error, and *errno* is set to one of the following values:

Value	Meaning
EACCES	The given pathname is not a directory, the directory is not empty, or the directory is the current working directory or root directory.
ENOENT	Pathname not found.

See Also

chdir(S), mkdir(DOS)

Example

```
#include <direct.h>

int result1, result2;

/* The following statements delete two directories:
** one at the root, and one in the current working
** directory.
*/

result1 = rmdir ("/data");
result2 = rmdir ("data");
```

Notes

This call must be compiled with the **-dos** flag.

Name

segread – command description

Syntax

#include <dos.h>

void segread (segregs);
struct SREGS *segregs;

Description

The *segread* function fills the structure pointed to by *segregs* with the current contents of the segment registers. The function is intended to be used with the *intdosx* and *int86x* functions to retrieve segment register values for later use.

Return Value

There is no return value.

See Also

intdosx(DOS), int86x(DOS), FP_SEG(DOS)

Example

```
#include <dos.h>

struct SREGS segregs;
unsigned int cs, ds, es, ss;

/* The following statements get the current values of
** the segment registers.
*/

segread (&segregs);
cs = segregs.cs;
ds = segregs.ds;
es = segregs.es;
ss = segregs.ss;
```

Notes

This call must be compiled with the **-dos** flag.

Name

setmode – Sets translation mode.

Syntax

```
#include <fcntl.h>
#include <io.h>

int setmode (handle, mode);
int handle;
int mode;
```

Description

The *setmode* function sets the translation mode of the file given by *handle* to *mode*. The *mode* must be one of the following manifest constants:

Manifest Constant	Meaning
O_TEXT	Set text (translated) mode. Carriage return/linefeed combinations (CR-LF) are translated into a single linefeed (LF) on input. Linefeed characters are translated into carriage return/linefeed combinations on output.
O_BINARY	Set binary (untranslated) mode. The above translations are suppressed.

setmode is typically used to modify the default translation mode of *stdin*, *stdout*, *stderr*, *stdaux*, and *stdprn*, but can be used on any file.

Return Value

If successful, *setmode* returns the previous translation mode. A return value of –1 indicates an error, and *errno* is set to one of the following values:

Value	Meaning
EBADF	Invalid file handle
EINVAL	Invalid *mode* argument (neither O_TEXT nor O_BINARY)

See Also

creat(S), fopen(S), open(S)

Example

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int result;

/* The following statement sets stdin to be binary
** (initially it is text).
*/

result = setmode (fileno (stdin),OBINARY);
```

Notes

This call must be compiled with the **-dos** flag.

Name

sopen – Opens a file for shared reading and writing.

Syntax

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <share.h>
#include <io.h>

int sopen (pathname, oflag, shflag[, pmode]);
char *pathname;
int oflag;
int shflag;
int pmode;
```

Description

The *sopen* function opens the file specified by *pathname* and prepares the file for subsequent shared reading or writing as defined by *oflag* and *shflag*. *oflag* is an integer expression formed by combining one or more of the following manifest constants, defined in *fcntl.h*. When more than one manifest constant is given, the constants are joined with the OR operator (|).

Oflag	**Meaning**
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file; this has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. Only applies when used with O_CREAT.
O_RDONLY	Open file for reading only; if this flag is given, neither O_RDWR nor O_WRONLY may be given.
O_RDWR	Open file for both reading and writing; if this flag is given, neither O_RDONLY nor O_WRONLY may be given.

O_TRUNC	Open and truncate an existing file to 0 length; the file must have write permission, and the contents of the file are destroyed.
O_WRONLY	Open file for writing only; if this flag is given, neither O_RDONLY nor O_RDWR may be given.
O_BINARY	Open file in binary (untranslated) mode. (See *fopen* for a description of binary mode.)
O_TEXT	Open file in text (translated) mode. (See *fopen* for a description of text mode.)

_TRUNC destroys the complete contents of an existing file. Use with care.

Shflag is a constant expression consisting of one of the following manifest constants, defined in **share.h.** See your MS-DOS documentation for detailed information on sharing modes.

Shflag	**Meaning**
SH_COMPAT	Set compatibility mode.
SH_DENYRW	Deny read and write access to file.
SH_DENYWR	Deny write access to file.
SH_DENYRD	Deny read access to file.
SH_DENYNONE	Permit read and write access.

The *pmode* argument is required only when _CREAT is specified. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. Otherwise, the *pmode* argument is ignored. The *pmode* argument is an integer expression containing one or both of the manifest constants S_IWRITE and S_IREAD, defined in **sys/stat.h.** When both constants are given, they are joined with the OR operator (|). The meaning of the *pmode* argument is as follows:

Value	**Meaning**
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD | S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read-only. Under MS-DOS all files are readable; it is not possible to give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE are equivalent.

sopen applies the current file permission mask to *pmode* before setting the permissions (see *umask*).

Return Value

sopen returns a file handle for the opened file. A return value of -1 indicates an error, and *errno* is set to one of the following values:

Value	Meaning
EACCES	Given pathname is a directory; or the file is read-only but an open for writing was attempted; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; MS-DOS versions 3.0 or later only).
EEXIST	The _CREAT and _EXCL flags are specified but the named file already exists.
EINVAL	SHARE.COM not installed.
EMFILE	No more file handles available (too many open files).
ENOENT	File or pathname not found.

See Also

close(S), creat(S), fopen(S), open(S), umask(S)

Example

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <share.h>
#include <io.h>

extern unsigned char _osmajor;
int fh;

    /* The _osmajor variable is used to test
    ** the MS-DOS version number before
    ** calling sopen.
    */

if (_osmajor >= 3)
    fh = sopen ("data", O_RDWR | O_BINARY, SH_DENYRW);
else
    fh = open ("data", O_RDWR | O_BINARY);
```

Notes

The *sopen* function should be used only under MS-DOS version 3.0 or later. Under earlier versions of MS-DOS, the *shflag* argument is ignored.

File sharing modes will not work correctly for buffered files, so do not use *fdopen* to associate a file opened for sharing (or locking) with a stream.

This call must be compiled with the **-dos** flag.

Name

spawnl, spawnvp – Creates a new process.

Syntax

```
#include <stdio.h>
#include <process.h>

int spawnl (modeflag, pathname, arg0, arg1...argn, NULL);

int spawnle (modeflag, pathname, arg0, arg1...argn, NULL, envp);

int spawnlp (modeflag, pathname, arg0, arg1...argn, NULL);

int spawnv (modeflag, pathname, argv);

int spawnve (modeflag, pathname, argv, envp);

int spawnvp (modeflag, pathname, argv);

int modeflag;
char *pathname;
char *arg0,*arg1...*argn;
char *argv [ ];
char *envp [ ];
```

Description

The *spawn* functions create and execute a new child process. There must be enough memory available for loading and executing the child process. The *modeflag* argument determines the action taken by the parent process before and during the *spawn*. The following values for *modeflag* are defined in **process.h**:

Value	Meaning
P_WAIT	Suspend parent process until execution of child process is complete
P_NOWAIT	Continue to execute parent process concurrently with child process
P_OVERLAY	Overlay parent process with child, destroying the parent (same effect as *exec* calls)

Only the P_WAIT and P_OVERLAY *modeflag* values may currently be used. The P_NOWAIT value is reserved for possible future implementation. An error value is returned if P_NOWAIT is used.

The *pathname* argument specifies the file to be executed as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *pathname* does not have a filename extension or end with a period (.), the spawn calls first append the extension .COM and search for the file; if unsuccessful, the extension .EXE is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the spawn calls search for *pathname* with no extension. The *spawnlp* and *spawnvp* routines search for *pathname* (using the same procedures) in the directories specified by the PATH environment variable.

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the *spawn* call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

The argument pointers may be passed as separate arguments (*spawnl*, *spawnle*, and *spawnlp*) or as an array of pointers (*spawnv*, *spawnve*, and *spawnvp*). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. (A different value will not produce an error.) Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* or *arg*[0] is not available for use in the child process. However, under MS-DOS 3.0 and later, the *pathname* is available as *arg0* or *arg*[0].

The *spawnl*, *spawnle* and *spawnlp* calls are typically used in cases where the number of arguments is known in advance. *arg0* is usually a pointer to *pathname*. *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a NULL pointer to mark the end of the argument list.

spawnv, *spawnve*, and *spawnvp* are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. *argv*[0] is usually a pointer to the *pathname*. *argv*[1] through *argv*[n] are pointers to the character strings forming the new argument list. *argv*[n+1] must be a NULL pointer to mark the end of the argument list.

Files that are open when a *spawn* call is made remain open in the child process. In the *spawnl*, *spawnlp*, *spawnv*, and *spawnvp* calls, the child process inherits the environment of the parent. *spawnle* and *spawnve* allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. *envp* is an array of character pointers, each element of which points to a null-terminated string defining an environment variable. Such a string has the form:

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotes.) When *envp* is NULL, the child process inherits the environment settings of the parent process.

Return Value

The return value is the exit status of the child process. The exit status is 0 if the process terminated normally. The exit status can also be set to a nonzero value if the child process specifically calls the *exit* routine with a nonzero argument. If not set, a positive exit status indicates an abnormal exit via an *abort* or an interrupt.

A return value of −1 indicates an error (the child process is not started), and *errno* is set to one of the following values:

Value	**Meaning**
E2BIG	The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K bytes.
EINVAL	Invalid *modeflag* argument.
ENOENT	File or pathname not found.
ENOEXEC	The specified file is not executable or has an invalid executable file format.
ENOMEM	Not enough memory is available to execute the child process.

See Also

abort(S), exec(S), exit(DOS)

Example

```
#include <stdio.h>
#include <process.h>

extern char **environ;

char *args[4];
int result;

args[0] = "child";
args[1] = "one";
args[2] = "two";
args[3] = NULL;
    .
    .
    .
/* All of the following statements attempt to spawn a
** process called "child.exe" and pass it 3 arguments.
** The first 3 suspend the parent, and the last 3
** overlay the parent with the child.
*/

result = spawnl (P_WAIT,"child.exe","child","one","two",
    NULL);
result = spawnle (P_WAIT,"child.exe","child","one",
    "two",NULL,environ);
result = spawnlp (P_WAIT,"child.exe","child","one",
    "two",NULL);
result = spawnv (P_OVERLAY,"child.exe",args);
result = spawnve (P_OVERLAY,"child.exe",args,environ);
result = spawnvp (P_OVERLAY,"child.exe",args);
```

Notes

The *spawn* calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the *setmode* routine should be used to set the translation mode of these files to the desired mode.

Signal settings are not preserved in child processes created by calls to *spawn* routines. The signal settings are reset to the default in the child process.

These calls must be compiled with the **-dos** flag.

Name

strlen – Returns the length of a string.

Syntax

#include <string.h>

int strlen (string);
char *string;

Description

The *strlen* function returns the length in bytes of *string* , not including the terminating null character ('\0').

Return Value

strlen returns the *string* length. There is no error return.

Example

```
#include <string.h>

char *string = "some space";
int result;
     .
     .
     .
/* Determine the length of a string.
*/

result = strlen (string); /* result = 10 */
```

Notes

This call must be compiled with the **-dos** flag.

Name

strlwr – Converts uppercase characters to lowercase characters.

Syntax

#include <string.h>

char *strlwr (string);
char *string;

Description

The *strlwr* function converts any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

Return Value

strlwr returns a pointer to the converted *string*. There is no error return.

See Also

strupr(DOS)

Example

```
#include <string.h>

char string[100], *copy;
    .
    .
    .
/* Make a copy of a string in lower case.
*/

copy = strlwr (strdup (string));
```

Notes

This call must be compiled with the **-dos** flag.

Name

strrev – Reverses the order of characters in a string.

Syntax

#include <string.h>

char *strrev (string);
char *string;

Description

The *strrev* function reverses the order of the characters in the given *string*. The terminating null character ('\0') remains in place.

Return Value

strrev returns a pointer to the altered *string*. There is no error return.

See Also

strcat(DOS), strset(DOS)

Example

```
#include <string.h>

char string[100];
int result;
     .
     .
     .
/* Determine if a string is a palindrome (the same
** string read forwards and backwards).
*/

result = strcmp (string,strrev (strdup (string)));

/* If result==0 the string is a palindrome.
*/
```

Notes

This call must be compiled with the **-dos** flag.

Name

strset – Sets all characters in a string to one charater.

Syntax

```
#include <string.h>

char *strset (string, c);
char *string;
char c;
```

Description

The *strset* function sets all characters of the given *string* except the terminating null character ('\0') to *c*.

Return Value

strset returns a pointer to the altered *string*. There is no error return.

See Also

string(S)

Example

```
#include <string.h>

char string[100], *result;
      .
      .
      .
/* Set a string to be all blanks.
*/

result = strset (string,' ');
```

Notes

This call must be compiled with the **-dos** flag.

Name

strupr – Converts lowercase characters to uppercase.

Syntax

```
#include <string.h>

char *strupr (string);
char *string;
```

Description

The *strupr* function converts any lowercase letters in the given *string* to uppercase. Other characters are not affected.

Return Value

strupr returns a pointer to the converted *string.* There is no error return.

See Also

strlwr(DOS)

Example

```
#include <string.h>

char string[100], *copy;
    .
    .
    .
/* The following statement makes a copy of a string in
** uppercase.
*/

copy = strupr (strdup (string));
```

Notes

This call must be compiled with the **-dos** flag.

Name

tell – Gets the current position of the file pointer.

Syntax

#include <io.h>

long tell (handle);
int handle;

Description

The *tell* function gets the current position of the file pointer (if any) associated with *handle*. The position is expressed as the number of bytes from the beginning of the file.

Return Value

tell returns the current position. A return value of −1L indicates an error, and *errno* is set to EBADF to indicate an invalid file handle argument. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

See Also

fseek(S), lseek(S)

Example

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

int fh;
long position;

fh = open ("data",ORDONLY);
    .
    .
    .
position = tell (fh); /* remember current position */
    .
    .
    .
lseek (fh, position, 0);     /* seek to previous position */
```

Notes

This call must be compiled with the **-dos** flag.

Name

ultoa – Converts numbers to characters.

Syntax

#include <stdlib.h>

char *ultoa (value, string, radix);
unsigned long value;
char *string;
int radix;

Description

The *ultoa* function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value*. It must be in the range 2–36.

Return Value

ultoa returns a pointer to *string*. There is no error return.

See Also

itoa(DOS), ltoa (DOS)

Example

```
#include <stdlib.h>

int radix = 16;
char buffer[40];
char *p;
    /* p will be "501d9138 */
p = ultoa (1344115000L,buffer,radix);
```

Notes

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

This call must be compiled with the **-dos** flag.

Name

ungetch – Returns a character to the console buffer.

Syntax

#include <conio.h>

int ungetch (c);
int c;

Description

The *ungetch* function pushes the character *c* back to the console, causing *c* to be the next character read. *ungetch* fails if it is called more than once before the next read.

Return Value

ungetch returns the character *c* if it is successful. A return value of EOF indicates an error.

See Also

cscanf(DOS), getch(DOS), getche(DOS)

Example

```
#include <conio.h>
#include <ctype.h>

char buffer[100];
int count = 0;
int ch;

/* The following code gets a token, delimited by blanks
** newlines, from the keyboard.
*/

ch = getche ( );

while (isspace (ch))        /* skip preceding white space */
    ch = getche ( );

while (count < 99) {        /* gather token */
        if (isspace (ch))   /* end of token */
         break;

    buffer[count++] = ch;
    ch = getche ( );
    }

ungetch (ch);               /* put back delimiter */
buffer[count] = '\0';       /* null terminate the token */
```

Notes

This call must be compiled with the **-dos** flag.

Permuted Index

Commands, System Calls, Library Routines and File Formats

This permuted index is derived from the "Name" description lines found on each reference manual page. Each *index* line shows the title of the entry to which the line refers, followed by the reference manual section letter where the page is found.

To use the *permuted index* search the middle column for a key word or phrase. The right hand column contains the name and section letter of the manual page that documents the key word or phrase. The left column contains additional useful information about the command. Commands or routines are also listed in the context of the *index* line, followed by a colon (:). This denotes the "beginning" of the sentence. Notice that in many cases, the lines wrap, starting in the middle column and ending in the left column. A slash (/) indicates that the description line is truncated.

06-21-87
SCO-514-210-024