Software Command Reference

IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal Computer Software

6138822

Software Command Reference

IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal Computer Software

First Edition (December 1984)

IBM Personal Computer XENIX Library Overview

The XENIX[1] System has three available products. They are the:

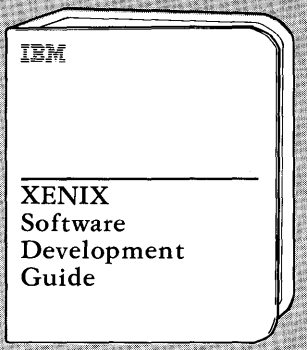
- Operating System
- Software Development System
- Text Formatting System

The following pages outline the XENIX Software Development System library.

[1] XENIX is a trademark of Microsoft Corporation.

XENIX Software Development System

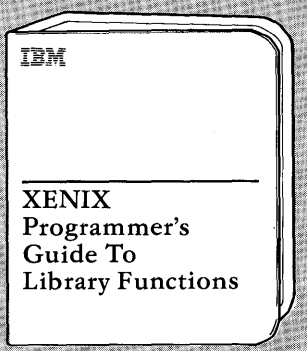
For C Language Users



- Creating language programs
- Invoking the C Compiler
- Program checkers, maintainers, and debuggers
- Using S-Files
- The C-Shell

A guide to the available programming tools in the XENIX environment.

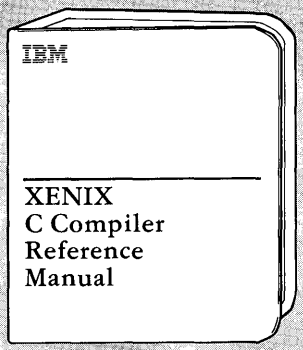
For All Users



- Using stream functions
- Screen processing
- Process controls
- Creating and using pipes
- Using signals and system resources

A reference to system calls, subroutines, and file formats. Use with the XENIX Software Command Reference.

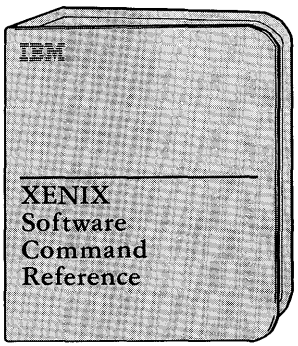
For Experienced Language Users



- Elements of the C programming language
- Preprocessor Directives
- Declarations
- Expressions and Assignments
- Description of functions and statements

A reference to the C programming language. Notational conventions are described throughout the manual.

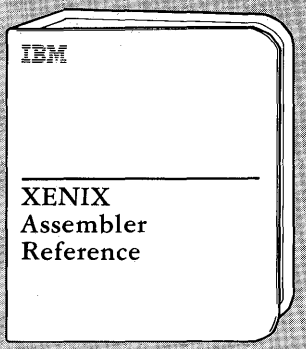
For All Users



- Software Development commands (CP)
- Command definition and syntax
- System calls and subroutines (S)
- System call and library function cross reference

A reference to Software Development System commands. Describes system services in the Operating System kernel.

For Assembler Users



- Assembly Language format
- Operands and Operators
- Directives
- Segment usage
- Machine instructions
- Assembler messages

A reference for programmers who use the IBM Personal Computer XENIX Assembler.

About This Book

This book describes the commands used in the IBM Personal Computer XENIX Software Development system, and the system services available in the operating system kernel. The commands used in the XENIX Software Development system are labled with the letters (CP) and listed in Section 1. The letter C stands for command and the letter P stands for programming. The system services, which include routines and system calls, are labeled with the letter (S). These commands are listed in Section 2. Appendix A is a system call and library function cross reference. Listed in this appendix are the functions found in various libraries, and the functions that directly invoke system primitives.

In references to other books, (C) stands for Command, (M) stands for Miscellaneous, and (F) stands for file format sections. These command are in the IBM Personal Computer *XENIX Command Reference*. References to (CT) stand for commands that come with the optional IBM Personal Computer XENIX Text Processing System. These commands are in Appendix A of the IBM Personal Computer *XENIX Text Formatting Guide*.

Related XENIX Publications

- IBM Personal Computer *XENIX Software Development Guide*
- IBM Personal Computer *XENIX Programmer's Guide to Library Functions*
- IBM Personal Computer *XENIX C Compiler Reference Manual*
- IBM Personal Computer *XENIX Assembler Reference*
- IBM Personal Computer *XENIX Installation Guide*
- IBM Personal Computer *XENIX Visual Shell*
- IBM Personal Computer *XENIX System Administration*
- IBM Personal Computer *XENIX Basic Operations Guide*
- IBM Personal Computer *XENIX Command Reference*

Contents

Section 1. Software Development Commands

Introduction to CP

This section describes the use of individual CP commands available in the Software Development System. Each command is labeled with the letters (CP) to distinguish it from commands available in the IBM Personal Computer *XENIX Command Reference* and IBM Personal Computer *XENIX Text Formatting Guide.*

The following example command outlines the format of this section. The **EXAMPLE**(CP) command is not a real XENIX command; it is only a sample of how the commands appear in this section.

EXAMPLE(CP)

Name

example - this is just an example of how this book is organized.

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

options 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 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.

Comments

Not all commands require options and arguments.

ADB(CP)

Name

adb - Invokes a general-purpose debugging program

Syntax

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

Description

Adb may be used to examine files and to provide a controlled environment for the execution of IBM Personal Computer XENIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if there is no symbol table, **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 written to the standard output. If the *-w* option is present, both *objfil* and *corefile* are created if necessary and opened for reading and writing so that files can be modified using **adb**. The Quit(Ctrl-\) and Interrupt(Del) 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, *dot* is set to *address*. Initially *dot* is set to 0. *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, addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see "Addresses." For most commands, *count* specifies how many times the command will be executed. The default *count* is 1.

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, 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 uppercase or lowercase 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 (in front of) 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 distinguish 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 refer to 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:

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 four 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, the match is for 4 bytes at a time instead of 2. If no match is found, *dot* is unchanged; otherwise *dot* is set to the matched location. If **mask** is omitted, -1 is used.

[?/]w value . . .
Writes the 2-byte **value** into the addressed location. If the command is **W,** writes 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, only the file position is changed. The **segnum** must be 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 is affected.

[?/]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 is created.

> 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, it is taken as the address of the current frame (instead of **bp).** If **C** is used, the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given, 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 nonzero variables in octal.

m Print the address map.

:modifier

Manage a subprocess. Available modifiers are:

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

dl Delete breakpoint at *address*.

r [arguments]
Run *objfil* as a subprocess. If *address* is given explicitly, the program is entered at this point; otherwise, the program is entered at its standard entry point. *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 upon 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, the subprocess is continued at this address. If no signal is specified, the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

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

k The current subprocess, if any, is terminated.

Variables

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, 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 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 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 that defines the size of the segment in memory at the current time. This size can change during execution.

When an 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 illegal.

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

/dev/mem
/dev/swap
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.

Comments

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.

ADMIN(CP)

Name

admin - Creates and administers SCCS files.

Syntax

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

Description

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 an 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 because 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. 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, 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 requires that they be created empty (no **-i** option). Note that the **-i** option implies the **-n** option.

-r*rel* The *rel* (release) 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).

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

-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 (that is, "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 (that is, "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 an irrecoverable 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 (for example, in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may 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 %Q% keyword in SCCS file text retrieved by **get**(CP).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the %M% 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 %Y% 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.

llist 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 who 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, 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.

-m[*mrlist*] The list of modification requests (SCCS) 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 SCCS numbers are validated if the **v** flag has a value (the name of an SCCS number validation program). Diagnostics will occur if the **v** flag is not set or SCCS validation fails.

-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 (that is, a new SCCS file is being created).

-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. 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 required to create a file. All writing done by **admin** is to a temporary x-file, called **x.file-name**, (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 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.

Warning: 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.file-name), 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.

AR(CP)

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. The *posname* is the name of a constituent file, and is required when certain keys are used. *Afile* is the archive file. The names are constituent files in the archive file. 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,** only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set **abi** is used, 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, 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)

Comments

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

AS(CP)

Name

as - XENIX 8086/186/286 Assembler.

Syntax

as [*options*] *source-file*

Description

As assembles 8086/186/286 assembly language source files and produces linkable object modules. The command accepts one or more *source-files,* and assembles each file separately. The source file names 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 Causes the assembler to output segments in alphabetic order. When omitted, segments are output in the order they occur.

-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 name is the same as the source filename with an extension of .lst.

Example
as -l foo.s *produces* foo.o and foo.lst

-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 upper case regardless of their original spelling.

-Mx Disables case sensitivity for local names and symbols only. This option is similar to the **-Mu** option, but does not affect names and symbols defined by the EXTRN and PUBLIC directives.

-n Suppresses the generation of symbol table output at the end of the listing. (Meaningful only when -l switch is used).

-o *outfile*
Causes the object output to be placed in the specified file. No default extension is assumed.

-O Causes values in the program listing to be displayed in octal. The default radix is hexadecimal. This option also applies to error messages.

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

Files

/bin/as

See Also

cc(C), ld(CP)
IBM Personal Computer *XENIX Assembler Reference*

Comments

Unless the **-r** is given, **as** 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.

Error messages for the XENIX assembler are listed in IBM Personal Computer *XENIX Assembler Reference.*

CB(CP)

Name

cb - Beautifies C programs.

Syntax

cb [*file*]

Description

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

CC(CP)

Name

cc - Invokes the C compiler.

Syntax

cc [*options*] *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" as 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 **as,** 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. The library is searched repeatedly to satisfy as many references as possible. 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 "main" program function.

There are the following options:

-C Preserves comments when preprocessing a file with **-E** or **-P**. That is, comments are not removed from the preprocessed source. This option may only be used with **-E** or **-P**.

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

-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 Creates an executable program for DOS. This program requires a .EXE extension.

-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. Default stack size if not given, is 4K-bytes.

-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. The 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.

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

-l*library*

Searches *library* for unresolved references to functions. The *library* must be an object file archive library in **ranlib** format.

-M *string*

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

s

Creates a small model program (default).

m

Creates a middle model program.

l

Creates a large model program.

e

Enables the far and near keywords.

2

Enables 80286 code generation for compiled C source files (default).

t *num* Sets the size of the largest data item in the data group to *num*. Default is 32,767.

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

-NGD *name*
Sets the data group name for each compiled or assembled source file to *name*. If not given, the name "DGROUP" is used.

-NGT *name*
Sets the text group name for each compiled or assembled source file to *name*. If not given, the name "IGROUP" is used.

-nl *num*
Sets the maximum length of external symbols to *num*. Names longer than *num* or 31 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.

-O Invokes the object code optimizer.

-o *filename*
Defines *filename* to be the name of the final executable program. This option overrides the default name a.out.

-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". Preprocessing performs the actions specified by the preprocessing directives.

-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 at normal termination of object program processing. This file can be examined using the **prof**(CP) command.

-S Creates an assembly source listing of the compiled C source file and copies this listing to the file whose basename is the same as the source but whose extension is ".s". This file is not suitable for assembly using **as**(CP). This option provides code for reading only.

-V *string*
Copies *string* to the object file created from the given source file. This option is often 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.

Many options (or equivalent forms of these options) are passed to the link editor as the last phase of compilation. 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 three different memory models: small, middle, and large. In addition, small model programs can be pure or impure.

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 128K bytes. 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. Procedural variables (function pointers) are 32 bits long in middle and large models. Programs using procedural variables must be carefully written. These procedural variables must be declared and used correctly. Text can be any size. Data must not exceed 64K bytes. Middle model 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.

Small, middle, and large model object files can only be linked with object and library files of the same model. It is not possible to combine small, middle, and large model object files in one executable program. Cc automatically selects the correct small,

middle, or large versions of the standard libraries based on the configuration option. It is up to the user to make sure that all of his own object files and private libraries are properly compiled in the appropriate model. Compilands are put into separate segments based on their source name. Thus, if you link two different compilands with the same source name both compilands will be put in the same segment. Each segment can be as large as 65k bytes. The upper limit on total code space is system dependent, but is usually greater than 150k bytes.

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

In both middle and large model programs, function pointers are 32 bits long. In large 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. **Lint**(CP) will help to check for correct use.

The **-NM, -NT, -ND, -NGT, -NGD** options may be used with middle and large 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.

Files

/bin/cc

See Also

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

Comments

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.

Error messages for the C Compiler are listed in IBM Personal Computer *XENIX C Compiler Reference Manual.*

CDC(CP)

Name

cdc - Changes the delta commentary of an SCCS delta.

Syntax

cdc -r*SID* [**-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)), 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). 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 to 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) that validates the correctness of the MR numbers. If a nonzero exit status is returned from the MR number validation program, **cdc** terminates and the delta commentary remains unchanged.

-y[*comment*]

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

COMB(CP)

Name

comb - Combines SCCS deltas.

Syntax

comb [**-o**] [**-s**] [**-p***sid*] [**-c***list*] *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 *SCCS IDentification* 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.

- s This argument causes **comb** to generate a shell procedure that produces a report for each file giving the filename, sizes (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

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

Diagnostics

Use **help**(CP) for explanations.

Comments

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.

CONFIG(CP)

name

config - configure a XENIX system

Syntax

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

Description

Config is a program that takes a description of a IBM Personal Computer XENIX system and generates a file that is a C program defining the configuration tables for the various devices on the system.

The **-t** option requests a short table of major device numbers for character and block type devices. This can facilitate the creation of special files.

> **Note:** The **-t** option does not know about devices that have aliases. However, the major-device numbers are always correct.

The **-c** option specifies the name of the configuration table file; **c.c** is the default name.

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

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

All configurations are assumed to have a set of required devices that must be present to run IBM Personal Computer XENIX, such as the system clock. These devices *must not* be specified in *dfile*.

First Part of dfile

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

devname number

where *devname* is the name of the device (as it appears in the **/etc/master** device table), and *number* is the number (decimal) of devices associated with the corresponding controller; *number* is optional, and if omitted, a default value which is the maximum value for that controller, is used.

Certain drivers may be provided with the system that are actually *pseudo-device* drivers; that is, there is no real hardware associated with the driver. Drivers of this type are identified in the book or keyed in.

Second Part of file.

The second part contains three types of lines. Note that *all* specifications of this part *are required,* although their order is arbitrary.

1. *Root/pipe device specification*

   Each line has three fields:

   ```
   root   devname   minor
   pipe   devname   minor
   ```

   where *minor* is the minor device number (in octal).

2. *Swap device specification*

   One line that contains five fields as follows:

   ```
   swap   devname   minor  swplo   nswap
   ```

where *swplo* is the lowest disk block (decimal) in the swap area and *nswap* is the number of disk blocks (decimal) in the swap area.

3. *Parameter specification*

One or more lines, each having two fields as follows:

```
name   number
```

where *name* is a tunable parameter name, and *number* is the desired value (in decimal) for the given parameter. Only names that have been defined in part 4 of the **/etc/master** file can be used; *number* overrides the default value for the given parameter. The following is a list of the available parameters:

buffers Maximum number of external (mapped-out) buffers available to the kernel. If set to 0, **config** computes the optimum number for the system.

sabufs Maximum number of internal (non-mapped) buffers available.

hashbuf Maximum number of hash buffers.

inodes Maximum number of inodes per file system.

files Maximum number of files per file system.

mounts Maximum number of mounted file systems.

coremap Maximum number of core map elements.

swapmap Maximum number of swap map elements.

pages Number of memory pages. On segmented systems such as the 286, this value should be 0.

calls Maximum number of entries in the system timeout table.

procs Maximum number of processes per system.

maxproc Maximum number of processes per user.

texts Maximum number of text segments per system.

clists Maximum number of clists per system.

locks Maximum number of file locks per system.

shdata Maximum number of shared data segments per system.

timezone Number of minutes difference between the local timezone and Greenwich Mean Time.

daylight Daylight savings time in effect (1), or not in effect (0).

cmask Default file creation mask for process 0.

maxprocmem
Maximum amount of memory available per process. This value cannot be greater than 75% of total user memory. If set to 0, **config** computes the optimum value.

Example

Suppose we wish to configure a system with the following devices:
one fixed disk drive controller with one driver
one diskette drive controller with one driver

We must also specify the following parameter information:
root device is a fixed disk (pseudo disk 3)
pipe device is a fixed disk (pseudo disk 3)
swap device is a fixed disk (pseudo disk 2)
with a swplo of 1 and an nswap of 2300
number of buffers is 50
number of processes is 50
maximum number of processes per user ID is 15
number of mounts is 8
number of inodes is 120
number of files is 120

number of calls is 30
number of texts is 35
number of character buffers is 150
number of swapmap entries is 50
number of memory pages is 512
number of file locks is 100
timezone is pacific time
daylight time is in effect

The actual system configuration would be specified as follows:

```
fixed disk   1
diskette     1
root   hd   3
pipe   hd   3
swap   hd   2     0   2300
* Comments may be inserted in this manner
buffers 50
procs    150
maxproc 15
mounts   8
inodes   120
files    120
calls    30
texts    35
clists   150
swapmap  50
pages (1024/2);
locks 100
timezone (8*60)
daylight 1
```

Files

/etc/master	default input master device table
c.c	default output configuration table file

See Also

master(F)

Diagnostics

Diagnostics are routed to the standard output and are self-explanatory.

CPP(CP)

Name

cpp - The C language preprocessor

Syntax

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

Description

Cpp is the C language preprocessor that 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. Therefore, 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 filenames as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. If the default is not supplied, it is standard input and standard output.

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 filename (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 tokens separated by commas, 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 only searched for in the standard places. See the **-I** option above for more detail.

#line *integer-constant "filename"*

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

#endif

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

#ifdef *name*

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

#ifndef *name*

The lines following do not appear in the output only 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 enviroment-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*

Lines following will appear in the output only if the *constant-expression* evaluates to nonzero. 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,** that 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 that 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 that matches this directive. So if lines previous to this directive are ignored, the following lines appear in the output (and the other way around).

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

Files

/usr/include 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.

Comments

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.

CREF(CP)

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)

Comments

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.

CSH(CP)

Name

csh - Invokes a shell command interpreter with C-like syntax.

Syntax

csh [**-cefinstvVxX**][*arg* . . .]

Description

Csh is a command language interpreter. It begins by executing commands from the file .cshrc in the home directory of the invoker. If this is a login shell, it also executes commands from the file .login there. In the normal case, the shell will begin reading commands from the terminal, prompting with %. Processing of arguments and the use of the shell to process files containing command scripts is described later.

The shell repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands from the file **.logout** in the user's home directory.

Lexical structure

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

In addition, strings enclosed in matched pairs of quotation marks, ' and ' or " and ", form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words.

These quotations have semantics to be described later. Within pairs of \ or " characters, a newline preceded by a \ gives a true newline character.

When the shell's input is not a terminal, the character # introduces a comment that continues to the end of the input line. It does not have this special meaning when preceded by \ and placed inside the quotation marks ' and ' or " and ".

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by | characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ; and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an &. Such a sequence is automatically prevented from being terminated by a hangup signal; the **nohup** command need not be used.

Any of the above may be placed in parentheses to form a simple command (which may be a component of a pipeline). It is also possible to separate pipelines with | | or && indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See "Expressions.")

Substitutions

The following sections describe the various transformations the shell performs on the input in the order in which they occur.

History Substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus, history substitutions provide a generalization of a *redo* function.

History substitutions begin with the character ! and may begin anywhere in the input stream if a history substitution is not already in progress. This ! may be preceded by a \ to override its special meaning; a ! is passed unchanged when it is followed by a

blank, tab, newline, =, or (. History substitutions also occur when an input line begins with ^. This special abbreviation will be described later.

Any input line that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal that consist of one or more words are saved on the history list, the size of which is controlled by the *history* variable. The previous command is always retained. Commands are numbered sequentially from 1.

For example, consider the following output from the history command:

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

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

With the current event 13, we can refer to previous events by event number !11, relatively as in !-2 (referring to the same event), by a prefix of a command word as in !d for event 12 or !w for event 9, or by a string contained in a word in the command as in !?mic? also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case !! refers to the previous command; thus !! alone is essentially a *redo*. The form !# refers to the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in !#:1.

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

0	First (command) word
n	**n**th argument
^	First argument, 1
$	Last argument
%	Word matched by (immediately preceding) ?**s**? search
x-y	Range of words
-y	Abbreviates 0- **y**
*	Abbreviates ^-$, or nothing if only one word in event
x*	Abbreviates **x**- $
x-	Like **x*** but omitting word $

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

h	Removes a trailing pathname component
r	Removes a trailing .xxx component (extracts the root of a filename)
e	extracts the extension of a filename
s/l/r/	Substitutes **l** for **r**
t	Removes all leading pathname components
&	Repeats the previous substitution
g	Applies the change globally, prefixing the above
p	Prints the new command but does not execute it
q	Quotes the substituted words, preventing substitutions

x Like q, but breaks into words at blanks, tabs, and newlines

Unless preceded by a **g**, the modification is applied only to the first modifiable word. In any case, it is an error for no word to be applicable.

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

A history reference may be given without an event specification, for example !$. In this case, the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus, !?foo?^!$ gives the first and last arguments from the command matching ?foo?.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ^. This is equivalent to !?s^, providing a convenient shorthand for substitutions on the text of the previous line. Thus ^lb^lib fixes the spelling of lib in the previous command. Finally, a history substitution may be surrounded with { and } if necessary to insulate it from the characters that follow. Thus, after ls -ld ~paul we might do !{l}a to do ls - ld ~paula, while !la would look for a command starting la.

Quotations With ' and "

The quotation of strings by ' and " can be used to prevent all or some of the remaining substitutions. Strings enclosed in quotes are prevented any further interpretation. Strings enclosed in quotes (") are variable and command expansion may occur.

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

Alias Substitution

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

Thus, if the alias for ls is ls - l the command "ls /usr" would map to "ls - l /usr". Similarly, if the alias for lookup was "grep !^ /etc/passwd" then "lookup bill" would map to "grep bill /etc/passwd".

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

Note that the mechanism allows aliases to introduce parser metasyntax. Thus, we can alias print 'pr \!* | lpr' to make a command that paginates its arguments to the line printer.

Variable Substitution

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

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the

verbose variable is a toggle that causes command input to be echoed. The setting of this variable results from the **-v** command line option.

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

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

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

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

The following sequences are provided for introducing variable values into the shell input. Except as noted, it is an error to refer to a variable that is not set.

$name
${name}

Are replaced by the words of the value of variable **name,** each separated by a blank. Braces insulate **name** from following characters that would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If **name** is not a shell variable, but is set in the environment, that value is returned (but : modifiers and the other forms given below are not available in this case).

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

May be used to select only some of the words from the value of **name.** The selector is subjected to $ substitution and may consist of a single number, or two numbers separated by a -. The first word of a variable's value is numbered 1. If the first number of a range is omitted, it defaults to 1. If the last member of a range is omitted, it defaults to $#name. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}

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

$0

Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}

Equivalent to $argv[number].

$*

Equivalent to $argv[*].

The modifiers :h, :t, :r, :q, and :x may be applied to the substitutions above as may :gh, :gt, and :gr. If braces { } appear in the command form, the modifiers must appear within the braces. Only one : modifier is allowed on each $ expansion.

The following substitutions may not be modified with : modifiers.

$?name
${?name}
Substitutes the string 1 if name is set, 0 if it is not.

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

$$
Substitutes the (decimal) process number of the (parent) shell.

Command and Filename Substitution

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

Command Substitution

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

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

Filename Substitution

If a word contains any of the characters *, ?, [or {, or begins with the character ~, then that word is a candidate for filename substitution. This word is then regarded as a pattern and is replaced with an alphabetically sorted list of filenames that match the pattern. In a list of words specifying filename substitution, it

is an error for no pattern to match an existing filename, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, the characters ~ and { being more akin to abbreviations.

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

The character ~ at the beginning of a filename is used to refer to home directories. Standing alone it expands to the invoker's home directory as reflected in the value of the variable *home*. When ~ is followed by a name consisting of letters, digits and - characters the shell searches for a user with that name and substitutes the home directory; thus ~ ken might expand to /usr/ken and ~ ken/chmach to /usr/ken/chmach. If the character ~ is followed by a character other than a letter or /, or appears not at the beginning of a word, it is left unchanged.

The metanotation a{b,c,d}e is a shorthand for abe ace ade. Left-to-right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus ~source/s1/{oldls,ls}.c expands to /usr/source/s1/oldls.c /usr/source/s1/ls.c, whether or not these files exist, without any chance of error if the home directory for source is /usr/source. Similarly, ../{memo,*box} might expand to ../memo ../box ../mbox. (Note that memo was not sorted with the results of matching *box.) As a special case {, } and {} are passed unchanged.

Input/Output

The standard input and standard output of a command may be redirected with the following syntax:

< name

Opens file **name** (which is first variable, command, and filename expanded) as the standard input.

<< word

Reads the shell input up to a line which is identical to **word. Word** is not subjected to variable, filename or command substitution, and each input line is compared to **word** before any substitutions are done on this input line. Unless a quoting backslash, double, or single quotation mark, or a back quotation mark appears in **word,** variable and command substitution is performed on the intervening lines, allowing \ to quote $, \ and '. Commands that are substituted have all blanks, tabs, and newlines preserved, except for the final newline, which is dropped. The resulting text is placed in an anonymous temporary file, which is given to the command as standard input.

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

The file **name** is used as standard output. If the file does not exist, it is created; if the file exists, it is truncated, and its previous contents are lost.

If the variable *noclobber* is set, the file must not already exist or it must be a character special file (for example a terminal or /dev/null) or an error results. This helps prevent accidental destruction of files. In this case, the ! forms can be used to suppress this check.

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

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

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

If a command is run detached (followed by &), the default standard input for the command is the empty file /dev/null. Otherwise, the command receives the environment in which the

shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form |& rather than just |.

Expressions

A number of the built-in commands (to be described later) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the **@, exit, if,**and **while** commands. The following operators are available:

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

Here the precedence increases to the right, with the operators:

```
== and !=
<=, >=, <, and >
<< and >>
+ and -
* / and %
```

forming groups at the same level. The == and != operators compare their arguments as strings; all others operate on numbers. Strings that begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions that are syntactically significant to the parser (& | < > ( )) they should be surrounded by spaces.

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

```
r       Read access
w       Write access
x       Execute access
e       Existence
o       Ownership
z       Zero size
f       Plain file
d       Directory
```

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

Control Flow

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

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

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto commands will succeed on nonseekable inputs.)

Built-In Commands

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

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

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

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

case label:
A label in a *switch* statement as discussed below.

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

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

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

echo wordlist

The specified words are written to the shell's standard output. An \c causes the echo to complete without printing a newline. An \n in **wordlist** causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw

See the description of the **foreach, if, switch,** and **while** statements below.

exec command

The specified command is executed in place of the current shell.

exit
exit (*expr*)

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

foreach *name (wordlist)*
...
end

The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching **end** are executed. (Both **foreach** and **end** must appear alone on separate lines.)

The built-in command **continue** may be used to continue the loop prematurely and the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with ? before any statements in the loop are executed.

glob *wordlist*

Like **echo**, but no \ escapes are recognized and words are delimited by null characters in the output. Useful for programs that wish to use the shell to filename expand a list of words.

goto *word*

The specified *word* is filename and command expanded to yield a string of the form label. The shell rewinds its input as much as possible and searches for a line of the form label: possibly preceded by blanks or tabs. Execution continues after the specified line.

history

Displays the history event list.

if (*expr*) **command**

If the specified expression evaluates true, the single **command** with arguments is executed. Variable substitution on **command** happens early, at the same time it does for the rest of the *if* command. **Command** must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is **not** executed.

if (*expr*) **then**
...
else if (*expr2*) **then**
...
else
...
endif

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

logout

Terminates a login shell. This is the only way to log out if *ignoreeof* is set.

nice
nice +*number*
nice *command*
nice + *number command*

The first form sets the **nice** for this shell to 4. The second form sets the **nice** to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using "nice -number . . . ." The command is always executed in a subshell, and the restrictions placed on commands in simple **if** statements apply.

nohup
nohup *command*

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. Unless the shell is running detached, **nohup** has no effect. All processes detached with & are automatically **nohup**ed. (Thus, **nohup** is not really needed.)

onintr
onintr -
onintr *label*

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

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

rehash

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories or if a systems programmer changes the contents of one of the system directories.

repeat *count command*

The specified *command* that is subject to the same restrictions as the *command* in the one-line *if* statement above, is executed *count* times. I/O redirection occurs exactly once, even if *count* is 0.

set
set *name*
set *name=word*
set *name[index]=word*
set *name=(wordlist)*

The first form of the command shows the value of all shell variables. Variables that have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word.* The fourth form sets the *index*th component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist.* In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv *name value*

Sets the value of the environment variable *name* to be *value,* a single string. Useful environment variables are TERM, the type of your terminal and SHELL, the shell you are using.

shift
shift *variable*

The members of *argv* are shifted to the left, discarding *argv[1].* It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source *name*

The shell reads commands from *name*. **Source** commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a **source** at any level terminates all nested **source** commands. Input during **source** commands is *never* placed on the history list.

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

Each case label is successively matched against the specified *string*, which is the first command and filename expanded. The file metacharacters *,?, and [. . .] may be used in the case labels, which are variable expanded. If none of the labels match before a default label is found, the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command **breaksw** causes execution to continue after the **endsw.** Otherwise, control may fall through case labels and default labels, as in C. If no label matches and there is no default, execution continues after the **endsw.**

time
time *command*

With no argument, a summary of time used by this shell and its children is printed. If arguments are given, the specified simple command is timed and a time summary as described under the **time** variable is printed. If necessary, an extra shell is created to print the time statistic when the command ends.

umask
umask *value*

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

unalias *pattern*
All aliases whose names match the specified pattern are discarded. Thus, all aliases are removed by unalias *. It is not an error if the pattern does not match any alias name.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unset *pattern*
All variables whose names match the specified pattern are removed. Thus, all variables are removed by unset *; with all variables removed, the program operates in an unpredictable manner. It is not an error if the pattern does not match any variable name.

wait
All child processes are waited for. If the shell is interactive, an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

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

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

Assignment operators, such as *= and +=, are available as in C. The space separating the name from the assignment operator is optional. Spaces are mandatory in separating components of *expr* that would otherwise be single words.

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

Predefined Variables

The following variables have special meaning to the shell. Of these, *argv, child, home, path, prompt, shell*, and *status* are always set by the shell. Except for *child* and *status*, this setting occurs only at initialization; these variables will not then be modified unless done explicitly by the user.

The shell copies the environment variable PATH into the variable *path* and copies the value back into the environment whenever *path* is set. Thus it is not necessary to worry about its setting other than in the file .cshrc, as inferior **csh** processes will import the definition of *path* from the environment.

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted, that is $1 is replaced by $argv[1], etc.

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

child The process number printed when the last command was forked with &. This variable is *unset* when this process terminates.

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

histchars Can be assigned a two-character string. The first character is used as a history character in place of !, the second character is used in place of the

substitution mechanism. For example, set histchars=,; will cause the history characters to be comma and semicolon.

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

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

ignoreeof If set, the shell ignores end-of-file from input devices that are terminals. This prevents a shell from accidentally being terminated by typing a Ctrl-D.

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

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

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

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

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

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

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

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

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set, but which are not executable by the system. (See the description of "Nonbuilt-In Command Execution" below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, 0200 is added to the status. Abnormal termination results in a core dump. Built-in commands that fail return exit status 1; all other built-in commands set status 0.

time Controls automatic timing of commands. If set, any command that takes more than this many cpu seconds causes a line giving user, system, and real

times and a utilization percentage that is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Nonbuilt-In Command Execution

When a command to be executed is found not to be a built-in command, the shell attempts to execute the command via **exec**(S). Each word in the variable *path* names a directory from which the shell attempts to execute the command. If it is given neither a **-c** nor a **-t** option, the shell will hash the names in these directories into an internal table so that it will only try an **exec** in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash),* or if the shell was given a **-c** or **-t** argument, and in any case for each directory component of *path* that does not begin with a /, the shell concatenates with the given command name to form a pathname of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus, (cd ; pwd) ; pwd prints the *home* directory; leaving you where you were (printing this after the home directory), while cd ; pwd leaves you in the home directory. Parenthesized commands are most often used to prevent *cd* from affecting the current shell.

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

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

Argument List Processing

If argument 0 to the shell is -, this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument, which must be present. Any remaining arguments are placed in *argv*.

-e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

-f The shell starts faster, because it will neither search for nor execute commands from the file .cshrc in the invoker's home directory.

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

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

-V Causes the *verbose* variable to be set even before .cshrc is executed.

-X Causes the *echo* variable to be set even before .cshrc is executed.

After processing of flag arguments, if arguments remain but none of the **-c, -i, -s,** or **-t** options were given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution

by $0. Since on a typical system most shell scripts are written for the standard shell (see **sh**(C)), the C shell will execute such a standard shell if the first character of a script is not a #, that is if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal Handling

The shell normally ignores *quit* signals. The *interrupt* and *quit* signals are ignored for an invoked command if the command is followed by &; otherwise, the signals have the values that the shell inherited from its parent. The shell's handling of interrupts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file .logout.

Files

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

Limitations

Words can be no longer than 512 characters. The number of arguments to a command that involves filename expansion is limited to 1/6 number of characters allowed in an argument list, which is 5120, less the characters in the environment. Also, command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

See Also

access(S), **exec**(S), **fork**(S), **pipe**(S), **signal**(S), **umask**(S), **wait**(S), **a.out**(F), **environ**(M)

Credit

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

Comments

Built-in control structure commands like **foreach** and **while** cannot be used with |, & or ;.

Commands within loops, prompted for by ?, are not placed in the *history* list.

It is not possible to use the colon (:) modifiers on the output of command substitutions.

Csh attempts to import and export the PATH variable for use with regular shell scripts. This only works for simple cases, where the PATH contains no command characters.

This version of **csh** does not support or use the process control features of the 4th Berkeley Distribution.

CTAGS(CP)

Name

ctags - Creates a tags file.

Syntax

ctags [**-u**][**-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 file 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 filename 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 offline 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:

-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.

> **Warning:** This option is implemented in a way that is rather slow; it is usually faster to simply rebuild the tags file.

-w Suppresses warning diagnostics.

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 if any, removed, 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.

DELTA(CP)

Name

delta - Makes a delta (list of changes needed to construct exactly one version of a file) to an SCCS file.

Syntax

delta [**-r***SID*] [**-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 generated file, or *g-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 on 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 that 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** 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) that will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, **delta** terminates (it is assumed that the MR numbers were not all valid).

-y[*comment*]

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

If **-y** is not specified and the standard input is a terminal, the prompt *comments?* is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped 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 6, "SCCS: A Source Code Control System" in the IBM Personal Computer *XENIX Software Development Guide.* The naming convention for these files is also described there.

g-file Existed before the execution of **delta;** removed after completion of **delta.**

p-file Existed before the execution of **delta ;** may exist after completion of **delta.**

q-file Created during the execution of **delta;** removed after completion of **delta.**

x-file Created during the execution of **delta;** renamed to SCCS file after completion of **delta.**

z-file Created during the execution of **delta;** removed during the execution of **delta.**

d-file Created during the execution of **delta;** removed after completion of **delta.**

/usr/bin/bdiff
Program to compute differences between the "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.

See Also

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

Diagnostics

Use **help**(CP) for explanations.

DOSLD(CP)

Name

dosld - XENIX to MS-DOS cross linker.

Syntax

dosld [*options*] *file* . . .

Description

dosld links the object file(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 option 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** uses 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.

-d Runtime debug information. This option instructs **dosld** to print information about what it is doing at runtime.

-m *filename*

Create map file. This option should be followed by a filename. **Dosld** creates a file with the given name in which it puts 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** uses 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** enters the given name into its symbol table as an undefined symbol. The **-u** option may appear more than once on the command line.

-w Windows option. This option instructs **dosld** to alter its normal behavior in the following ways: (1) combine all code segments together into a single code segment; (2) replace all long calls with short calls; and (3) replace all long jumps with short jumps.

-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 passed 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)

GET(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," below).

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 *SCCS IDentification* 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 two-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 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 **get**s 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 Chapter 6, "SCCS: A Source Code Control System," in the IBM Personal Computer XENIX *Software Development 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, 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, irrecoverable 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 (for example, **- r1),** or release and level (for example, **- 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, because the SID of the delta to be created may not be what you 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 four-character string **@(#)** recognizable by **what**(C).
%W%	A shorthand notation for constructing **what**(C) strings for IBM Personal Computer XENIX program files. %W% = %Z%%M%<horizontal-tab>%I%

Keyword	Value
%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, for the file **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 blank character if the delta was applied; * otherwise.
- A blank character if the delta was applied or wasn't applied and ignored; * if the delta wasn't applied and wasn't ignored.
- 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).

- Blank.
- SCCS identification (SID).
- Tab character.
- Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
- Blank.
- 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 (two-bytes) process ID of the command (that is, **get**) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of **get**. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

See Also

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

Diagnostics

Use **help**(CP) for explanations.

Comments

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

GETS(CP)

Name

gets - Gets a string from the standard input.

Syntax

gets [*string*]

Description

Gets can be used with **csh**(CP) 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**(CP)

HDR(CP)

Name

hdr - Displays selected parts of object files.

Syntax

hdr [**-dhprsSt**] *file* . . .

Description

Hdr displays object 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 object 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 that 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 (twobytes), or long. The last field indicates, 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 object 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)

HELP(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** prompts for one.

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

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

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

type 3 — Is all numeric (for example, 212)

The response of the program is 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.

LD(CP)

Name

ld - Invokes the link editor.

Syntax

ld [*options*] *filename* . . .

Description

Ld is the IBM Personal Computer XENIX link editor. It creates an executable program by combining one or more object files and copying the executable result to the file a.out. The *filename* must name an object or library file. These names must 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 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.

The following options are:

-A*num*
Creates a stand-alone 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 stand-alone programs.

-F*num*
Sets the size of the program stack to *num* bytes. Default stack size if not given, is 2K bytes.

-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.

-Ms Creates small model program and checks for error, such as fixup overflow. This option is reserved for object files compiled or assembled using the small model configuration. This is the default model if no **-M** option is given.

-Mm Creates middle model program and checks for errors. This option is reserved for object files compiled or assembled using the middle model configuration. This option implies **-i.**

-Ml Creates a large model program and checks for errors. The option is reserved for object files compiled using the large model configuration. This option implies **-i.**

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

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 because failure to give command line arguments in the correct order can result in errors.

Files

/bin/ld

See Also

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

Comments

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.

Error messages for the XENIX Software Development System are listed in IBM Personal Computer *XENIX Software Development Guide*.

LEX(CP)

Name

lex - Generates programs for lexical analysis.

Syntax

lex [-ctvn] [*file*] . . .

Description

Lex generates programs to be used in simple lexical analysis of text. A file lex.yy.c is generated which, when loaded with the **lex** library, copies the input to the output except when a string specified in the file is found. If a string is found, the corresponding program text is executed.

The input *file* contains strings and expressions to be searched for, and C text to be executed when strings are found. Multiple files are treated as a single file. If no files are specified, standard input is used.

The options must appear before any files. The options are:

-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 Suppresses the - summary.

Strings and Operators

Lex strings may contain square brackets to indicate character classes, as in **[abx-z]** to indicate **a, b, x, y, and z**. 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. Thus, **[a-zA-Z]+** matches a string of letters. 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 \ .

Routines and Variables

Matching is done in order of the strings in the file. The actual string matched is left in *yytext,* an external character array. Three subroutines defined as macros are expected: **input()** to read a character; **unput(c)** to replace a character read; and **output(c)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex(),** and the library contains a **main()**, which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match to be 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. The external names generated by **lex** all begin with the prefix **yy or YY.**

Lex File Format

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**(CP). Lines preceding %% that 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 braces do not imply parentheses; only string substitution is done.

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

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

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

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

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

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

Example

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

See Also

yacc(CP)
IBM Personal Computer *XENIX Software Development Guide*

Comments

This program translates its input into C source code, which in segmented programming environments, is suitable for compiling as a small model program only (see **cc**(CP)).

LINT(CP)

Name

lint - Checks C language usage and syntax.

Syntax

lint [-abchlnpuvx] *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 that 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 that 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:

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

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

-llibname
Checks functions 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 change the behavior of **lint**. These are:

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

/*VARARGSn */
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 printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed 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)

Comments

Exit(S), and other functions that do not return, are not understood. This can cause incorrect error messages.

LORDER(CP)

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 second. 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)

Comments

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.

M4(CP)

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:

-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 4096.

-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 that 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 (that is, ˇ).

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 as the second, 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 is 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.**

MAKE(CP)

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

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* will override built-in rules.

-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 continues 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.

-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.

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

.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(Ctrl-\) or Interrupt(Del) keys are pressed.

.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, the command was null.

INTERRUPT and QUIT cause the target to be deleted unless the target depends on the special name **.PRECIOUS**.

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

Five internally maintained macros 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 that is out of date with respect to the target (that is, 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 uppercase **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 (that is **.c**:) is the definition of how to build **x** from **x.c**. In effect, the other suffix is null. This is useful for building targets from only one source file (for example, 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 displays 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 *$(LIB)(file.o)* both refer to an archive library that contains *file.o*. (This assumes the **LIB** macro has been previously defined.) The expression *$(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. 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 unnecessary 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**. (One cannot 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)

Comments

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.

MKSTR(CP)

Name

mkstr - Creates an error message file from C source.

Syntax

mkstr [-] *messagefile prefix file . . .*

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

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** focuses 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 an *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)
{
   char buf[256];

   if (efil < 0) {
         efil = open(efilname, 0);
         if (efil < 0) {
               oops:
               perror(efilname);
               exit();
         }
   }
   if (lseek(efil, (long) a1, 0) | | read(efil, buf, 256) <= 0)
         goto oops;
   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.

NM(CP)

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 is 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), or **K** (8086 common segment). If the symbol table is in segmented format, symbol values are displayed as **segment:offset**. If the symbol is local (nonexternal) the type letter is in lowercase. The output is sorted alphabetically.

Options are:

-a Print only absolute symbols.

-c Print only C program symbols (symbols that 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 option implies the **-i** and **-n** options.

-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.

-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)

PROF(CP)

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)

Comments

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.

PRS(CP)

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 XENIX 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 *SCCS IDentification* (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, that is, delta type = *R,* (see **rmdel**(CP)) and existing, that is, 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 newline.

User-supplied text is any text other than recognized data keywords. A tab is specified by \t and 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:	SSCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S

TABLE 1. SCCS Files Data Keywords (Continued)

Keyword	*Data Item*	*File Section*	*Value*	*Format*
: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 name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(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:

The following example:

```
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
```

The following:

```
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.

RANLIB(CP)

Name

ranlib - Converts archives to random libraries.

Syntax

ranlib *arch1 arch2 . . .*

Description

Ranlib converts each *archive* to a form that can be utilized more rapidly by the linker, by adding a table of contents named __ __.SYMDEF to the beginning of the archive.

See Also

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

Comments

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

RATFOR(CP)

Name

ratfor - Converts rational FORTRAN into standard FORTRAN

Syntax

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

Description

Ratfor converts a rational dialect of FORTRAN into standard 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]
```

and 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 option **-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; the option **-6x** makes the continuation character **x** and places it in column 6.

Comments

This program translates its input into C source code, which in segmented programming environments, is suitable for compiling as a small model program only (see **cc**(CP)).

REGCMP(CP)

Name

regcmp - Compiles regular expressions.

Syntax

regcmp [-] *file*

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 is 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 in C programs, or *file.c* files may be compiled and later loaded. In the C program that 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)
```

applies the regular expression named **telno** to **line**.

See Also

regex(S)

Comments

This program translates its input into C source code, which, in segmented programming environments, is suitable for compiling as a small model program only (see **cc**(CP)).

RMDEL(CP)

Name

rmdel - Removes a delta from an SCCS file.

Syntax

rmdel -r*SID 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-filec* (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.

SACT(CP)

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 that is, executed a **get** for editing.

Field 4 Contains the date that **get -e** was executed.

Field 5 Contains the time that **get -e** was executed.

See Also

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

Diagnostics

Use **help**(CP) for explanations.

SCCSDIFF(CP)

Name

sccsdiff - Compares two versions of an SCCS file.

Syntax

sccsdiff **-r***SID1* **-r***SID2* [**-p**] [**-s***n*] *files*

Description

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

-r*SID?* *SID1 and SID2 specify* the deltas of an SCCS file that are to be compared. Versions are passed to **bdiff**(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.

SIZE(CP)

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)

SPLINE(CP)

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, that is 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.

Comments

A limit of 1000 input points is silently enforced.

STACKUSE(CP)

Name

stackuse - Determines stack requirements for C programs.

Syntax

stackuse [**-m** *startsym*] [**-r** *fakeref*] [**-s** *libstack*] [*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*
: Print only the specified start ("main") symbol. If this option is not specified all start symbols (those that 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 metasubroutine meaning all undefined routines.

-a Print 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 that begin with a pound sign (#) are treated as comments and otherwise ignored.

Diagnostics

Usage (fatal).

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

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

Files

/usr/lib/stackuse/*	Passes, libraries
/tmp/*	Temporary Y files used by passes.

Comments

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

STRINGS(CP)

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, each string is preceded by its decimal offset in the file. If the - *number* flag is given, *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.

STRIP(CP)

Name

strip - Removes symbols and relocation bits.

Syntax

strip [**-dehrsStx**]*file* . . .

Description

Strip removes selected parts of an object file, including the header, text, data, relocation records, and symbol table. Strip works directly on the named files; nothing is written to the standard output.

Strip is typically used to remove symbol table and relocation information from a file after debugging has been completed. It also is useful for creating a compact namelist file in which text and data have been removed.

-d Strip data and the data relocation records.

-e Strip the extended header.

-h Strip the header and extended header.

-r Strip all relocation records except the x.out short form.

-s Strip the symbol table.

-S Strip the segment table.

-t Strip text and the text relocation records.

-x Strip all relocation records.

Strip has the same effect as the **-s** option of **ld.** If no options are given, the **-r** and **-s** options are implied.

Although **strip** can be used to remove an x.out header from an 80286 relocatable file, it cannot be used to remove run-time relocation records.

Files

/tmp/s* Temporary file

See Also

ld(CP), **a.out**(F)

TIME(CP)

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 output.

See Also

times(S)

TSORT(CP)

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: An odd number of fields is in the input file.

Comments

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

UNGET(CP)

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 before 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 ambiguous, or if it is necessary and omitted on the command line.

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

-n Causes the retention of the file that would normally be removed from the current directory.

See Also

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

Diagnostics

Use **help**(CP) for explanations.

VAL(CP)

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 eight-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 (SCCS IDentification String) is an SCCS delta number. A check is made to determine if the SID is ambiguous (For example **rl** is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (For

example **r 1.0** or **r 1.1.0** are invalid because neither case can exist as a valid delta number). If the SID is valid and not ambiguous, a check is made to determine if it actually exists.

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

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

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

bit 0 = Missing file argument.

bit 1 = Unknown or duplicate 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 lines (when reading the standard input). In these cases, an aggregate code is returned, a logical OR of the codes generated for each command line and file processed.

See Also

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

Diagnostics

Use **help**(CP) for explanations.

Comments

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

XREF(CP)

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)

XSTR(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
```

extracts the strings from the C source file 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 be compiled. The strings from this file are placed in the strings data base if they are not there already. Repeated strings and strings that 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

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 that contains strings that 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* Temporary 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.

Comments

If a string is a suffix of another string in the data base, but the shorter string is seen first by **xstr,** both strings are placed in the data base when just placing the longer one there will do.

YACC(CP)

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)

Comments

Because filenames are fixed, at most one **yacc** process can be active in a given directory at a time.

This program translates its input into C source code, which in segmented programming environments, is suitable for compiling as a small model program only (see **cc**(CP)).

Section 2. System Calls and Subroutines

Introduction to (S)

This section describes all the system services. System services include all routines or system calls that are available in the operating system kernel. These services are labled with the letter (S). These routines are available to a C program automatically as part of the standard library *libc*. A **Synopsis** listing the function's name, type, arguments, and declarations of the arguments, is given for each system call and subroutine. Other routines are available in a variety of libraries. On 8086/88 and 80286 systems, versions for small, middle, and large model programs are provided (that is, three of each library).

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 -lname to the compiler or linker, where *name* is the name listed below. For example **-lm,** and **-ltermcap** 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.

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 that 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 5120 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 that 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, for example, in **mount.**

16 EBUSY Device busy
An attempt was made 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 also occurs 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, for example, **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; for example, 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
Some invalid argument (for example, 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 20 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that 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.

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; that is, 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, that is, without issuing a **waitsem** on the semaphore.

39 EISNAM A name file
A name file (semaphore, shared data, etc.) was specified when not expected.

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).

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 IDS. 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 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's 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 associated with it a concept of a root directory and a current working directory for the purpose of resolving pathname searches. A process's 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 is true:

The process's effective user ID is super-user.

The process's 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's effective user ID does not match the user ID of the owner of the file, and the process's 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's effective user ID does not match the user ID of the owner of the file, and the process's 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).

See Also

intro(C)

A64L(S)

Name

a64l, l64a - Converts between long integer and base 64 ASCII.

Synopsis

```
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:

.	0
/	1
0-9	2-11,
A-Z	12-37
a-z	38-63.

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.

Comments

The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call.

ABORT(S)

Name

abort - Generates an IOT fault.

Synopsis

```
abort ( )
```

Description

Abort 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".

ABS(S)

Name

abs - Returns an integer absolute value.

Synopsis

```
int abs (i)
int i;
```

Description

Abs returns the absolute value of its integer operand.

See Also

fabs in **floor**(S)

Comments

If the largest negative integer supported by the hardware is given, the function returns it unchanged.

ACCESS(S)

Name

access - Determines accessibility of a file.

Synopsis

```
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 is 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's 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)

Comments

The super-user (root) may access any file, regardless of permission settings.

ACCT(S)

Name

acct - Enables or disables process accounting.

Synopsis

```
int acct (path)
char *path;
```

Description

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record is 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 that 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 is 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

On 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)

ALARM(S)

Name

alarm - Sets a process's alarm clock.

Synopsis

```
unsigned alarm (sec)
unsigned sec;
```

Description

Alarm sets the calling process's alarm clock to **sec** seconds. After **sec** "real-time" seconds have elapsed, 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's alarm clock.

If **sec** is 0, any previously made alarm request is canceled.

Return Value

Alarm returns the amount of time previously remaining in the calling process's alarm clock.

See Also

pause(S), **signal**(S)

ASSERT(S)

Name

assert - Helps verify validity of program.

Synopsis

```
#include <assert.h>

assert (expression);
```

Description

This macro is useful for putting diagnostics into programs under development. When it is executed, if **expression** is false, it prints

Assertion failed: file *name*, line *nnn*

on the standard error file and exits. *Name* is the source filename and *nnn* the source line number of the **assert** statement.

Comments

To suppress calls to **assert,** use the option "-DNDEBUG" when compiling the program; see **cc**(CP)).

ATOF(S)

Name

atof, atoi, atol - Converts ASCII to numbers.

Synopsis

```
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)

Comments

There are no provisions for overflow.

BESSEL(S)

Name

j0, j1, jn, y0, y1, yn - Performs Bessel functions.

Synopsis

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
init n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

Description

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

Comments

Negative arguments cause **y0, y1,** and **yn** to return a huge negative value.

BSEARCH(S)

Name

bsearch - Performs a binary search.

Synopsis

```
char *bsearch (key, base, nel, width, compar)
char *key;
char *base;
int 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. 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 is the number of elements in the table. The fourth is the width of an element in bytes. The last argument is the name of the comparison routine. It is called with two arguments that are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

Return Value

If the key cannot be found in the table, a value of 0 is returned.

See Also

lsearch(S), **qsort**(S)

CHDIR(S)

Name

chdir - Changes the working directory.

Synopsis

```
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 is 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's allocated address space. [EFAULT]

Return Value

On 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)

CHMOD(S)

Name

chmod - Changes mode of a file.

Synopsis

```
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 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, mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user finishes with 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 is 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's allocated address space. [EFAULT]

Return Value

On 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)

CHOWN(S)

Name

chown - Changes the owner and group of a file.

Synopsis

```
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, are cleared.

Chown fails and the owner and group of the named file remains unchanged if one or more of the following is 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's allocated address space. [EFAULT]

Return Value

On 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)

CHROOT(S)

Name

chroot - Changes the root directory.

Synopsis

```
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 /.

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 fails and the root directory remains unchanged if one or more of the following is 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's allocated address space. [EFAULT]

Return Value

On 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)

CHSIZE(S)

Name

chsize - Changes the size of a file.

Synopsis

```
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, 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

On 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)

Comments

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.

CLOSE(S)

Name

close - Closes a file descriptor.

Synopsis

```
int close (fildes)
int fildes;
```

Description

Fildes is a file descriptor obtained from a **creat, open, dup, fcntl,** or **pipe** system call. **Close** closes the file descriptor indicated by **fildes.**

Close will fail if **fildes** is not a valid open file descriptor. [EBADF]

Return Value

On 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)

CONV(S)

Name

toupper, tolower, __toupper, __tolower, toascii - Translates characters.

Synopsis

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int __toupper (c)
int c;

int __tolower (c)
int c;

int toascii (c)
int c;
```

Description

Toupper and **tolower** 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)

Comments

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.

CREAT(S)

Name

creat - Creates a new file or rewrites an existing one.

Synopsis

```
int creat (path, mode)
char *path;
int mode;
```

Description

Creat creates a new ordinary file or prepares to rewrite an existing file named by the 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's effective user ID, the file's group ID is set to the process's effective group ID, and the access permission bits (that is, 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** then modifies the access permission bits as follows:

> All bits set in the process's file mode creation mask are cleared. See **umask**(S).
>
> The "save text image after execution bit" is cleared. See **chmod**(S).

On successful completion, a nonnegative 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 20 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 is 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]

Twenty file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

Return Value

On 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)

Comments

Open(S) is preferred to **creat.**

CREATSEM(S)

Name

creatsem - Creates an instance of a binary semaphore.

Synopsis

```
sem_num = creatsem (sem_name,mode);
int sem_num,mode;
char *sem_name;
```

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.**

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.

Comments

After a **creatsem** you must do a **waitsem** to gain control of a given resource.

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX [1] versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

[1] UNIX is a trademark of AT & T Technology Inc.

CTERMID(S)

Name

ctermid - Generates a filename for a terminal.

Synopsis

```
#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, **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>**.

Comments

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 refers 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)

CTIME(S)

Name

ctime, localtime, gmtime, asctime, tzset - Converts date and time to ASCII.

Synopsis

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

tzset ( )

extern long timezone;
extern int daylight;
extern char tzname;
```

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\Ø

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), and a flag that is nonzero if daylight saving time is in effect. **Localtime** corrects for the time zone and possible daylight saving 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 (for example, in Eastern Standard Time (EST), *timezone* is 5*60*60); the external integer variable *daylight* is nonzero if and only if the standard U.S.A. Daylight Saving 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. The value of TZ must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich time in hours, followed by an optional three-letter name for a daylight time zone. The difference can be a negative number if the current location is east of England. For example, the setting for New Jersey would be EST5EDT. The effects of setting TZ are thus to change the values of the external variables *timezone* 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 function **tzset** sets the external variables from TZ ; it is called by **asctime** and may also be called explicitly by the user.

See Also

time(S), **getenv**(S), **environ**(M)

Comments

The return values point to static data whose contents are overwritten by each call.

CTYPE(S)

Name

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii - Classifies characters.

Synopsis

```
#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 hexadecimal 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.

See Also

ascii(M)

CURSES(S)

Name

curses - Performs screen and cursor functions.

Synopsis

```
#include <curses.h>
WINDOW *curscr, *stdscr;
```

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, **curscr,** and 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. 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 loader options **-lcurses** and **-ltermcap.**

See Also

termcap(M), **stty**(C), **setenv**(S)
IBM Personal Computer *XENIX Programmer's Guide to Library Functions*

Functions

> **Note:** Some of the following functions use y,x coordinates. The y variable indicates row and the x variable indicates column.

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 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 x,y;
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, begin_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 restty()
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 wdeleteln(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)
IBM Personal Computer *XENIX Programmer's Guide to Library Functions*

Credit

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

CUSERID(S)

Name

cuserid - Gets the login name of the user.

Synopsis

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

Description

Cuserid returns a pointer to string that 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)

Comments

Cuserid uses **getpwnam** (see **getpwent**(S)); thus the results of a user's call to the latter is obliterated by a subsequent call to the former.

DBM(S)

Name

dbminit, fetch, store, delete, firstkey, nextkey - Performs database functions.

Synopsis

```
typedef struct { char *dptr;int dsize; } datum;

dbminit (file)
char *file;

datum fetch (key)
datum key;

store (key, content)
datum key, content;

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 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, is 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** returns 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 O.K. Routines that return a **datum** indicate errors with a null (0) **dptr.**

Comments

The .pag file will contain holes, so that its apparent size is about four times its actual content. Older UNIX 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, which 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 returns an error if block off 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.

DEFOPEN(S)

Name

defopen, defread - Reads default entries.

Synopsis

```
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.

DUP(S)

Name

dup, dup2 - Duplicates an open file descriptor.

Synopsis

```
int dup (fildes)
int fildes;

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 (that is, 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 is true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty file descriptors are currently open. [EMFILE]

Return Value

On 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

creat(S), **close**(S), **exec**(S), **fcntl**(S), **open**(S), **pipe**(S)

Comments

Dup2 is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. **Dup2** must be linked with the compiler/linker option -lx.

ECVT(S)

Name

ecvt, fcvt, gcvt - Performs output conversions.

Synopsis

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

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)

Comments

The return values point to static data whose content is overwritten by each call.

END(S)

Name

end, etext, edata - Last locations in program.

Synopsis

```
extern end;
extern etext;
extern edata;
```

Description

These names do not refer to either routines or locations with interesting contents. The address of **etext** is the first address above the program text, **edata** above the initialized data region, and **end** above the uninitialized data region.

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, you can not assume 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.

EXEC(S)

Name

execl, execv, execle, execve, execlp, execvp - Executes a file.

Synopsis

```
int execl (path, arg0, arg1, . . . , argn, 0)
char *path, *arg0, *arg1, . . . , *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0,arg1, . . . , argn, 0, envp)
char *path, *arg0, *arg1, . . . , *argn,*envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, . . . , argn, 0)
char *file, *arg0, *arg1, . . . , *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

Description

Exec in all its forms 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 are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the 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

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** 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 1 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.

The following are example calls to each **exec**:

```
execl("/bin/sh", "sh", "-c", argv[1],0);
execle("/bin/sh", "-c", argv[1],0, environ);
execlp("ls", "ls", (char*)0);
execv("/bin/cat", argv);
execve("/bin/cat", argv, environ);
execvp( argv[1], &argv[1]);
```

Exec will fail and return to the calling process if one or more of the following is 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 allowed by the memory model used or the system-imposed maximum. [ENOMEM]

The number of bytes in the new process's 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** points 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)

EXIT(S)

Name

exit - Terminates a process.

Synopsis

```
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's termination and the low-order 8 bits (that is, bits 0377) of **status** are made available to it; see **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's existing child processes and zombie processes is set to 1. This means the initialization process (see **introduction**(S)) inherits each of these processes.

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 processes that has a process group ID equal to that of the calling process.

See Also

signal(S), **wait**(S)

Warning

See **Warning** in **signal**(S)

EXP(S)

Name

exp, log, pow, sqrt, log10 - Performs exponential, logarithm, power, square root functions.

Synopsis

```
#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. A truly outrageous argument may also result in *errno* being set to ERANGE. **Log** returns a huge negative value and sets *errno* to EDOM when **x** is nonpositive. **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.

FCLOSE(S)

Name

fclose, fflush - Closes or flushes a stream.

Synopsis

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

Description

Fclose causes any buffers for the name **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 are detected.

See Also

close(S), **fopen**(S), **setbuf**(S)

FCNTL(S)

Name

fcntl - Controls open Files.

Synopsis

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;
```

Description

Fcntl provides for control over open files. **Fildes** is an open file descriptor obtained from a **creat, open, dup, fcntl,** or **pipe** system call.

The **cmds** 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 (that is, both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

Same file status flags (that is, both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across **exec**(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 remains open across **exec**; otherwise, the file is 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 **1** as above).

F_GETFL Gets file status flags.

F_SETFL Sets file status flags to **arg.** Only certain flags can be set.

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 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and **arg** is negative or greater than 20. [EINVAL]

Return Value

On 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

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

close(S), **exec**(S), **open**(S)

FERROR(S)

Name

ferror, feof, clearerr, fileno - Determines stream status.

Synopsis

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

Description

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.

Feof returns nonzero when end-of-file is read on the named input **stream,** otherwise zero.

Clearerr resets the error indication on the named **stream.**

Fileno returns the integer file descriptor associated with the **stream**; see **open**(S).

Ferror, feof, and **fileno** are implemented as macros; they cannot be redeclared.

See Also

open(S), **fopen**(S)

FLOOR(S)

Name

floor, fabs, ceil, fmod - Performs absolute value, floor, ceiling and remainder functions.

Synopsis

```
#include <math.h>

double floor (x)
double x;

double fabs (x)
double x;

double ceil (x)
double x;

double fmod (x,y)
double x, y;
```

Description

Floor returns the largest integer (as a double precision number) not greater than **x.**

Fabs returns | 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)

FOPEN(S)

Name

fopen, freopen, fdopen - Opens a stream.

Synopsis

```
#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 that encounters the end of the file.

See Also

open(S), **fclose**(S)

Diagnostics

Fopen and **freopen** return the pointer NULL if **filename** cannot be accessed.

FORK(S)

Name

fork - Creates a new process.

Synopsis

```
int fork ()
```

Description

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

The child process has a unique process ID.

The child process has a different parent process ID (that is, 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.

The child process's *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 is 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

On 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), **wait**(S)

FREAD(S)

Name

fread, fwrite - Performs buffered binary input and output.

Synopsis

```
#include <stdio.h>

int fread ((char *) ptr, sizeof (*ptr), nitems,stream)
int nitems;
FILE *stream;

int fwrite ((char *) ptr, sizeof (*ptr), nitems,stream)
int 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**. 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.** It returns the number of items actually written.

See Also

read(S), **write**(S), **fopen**(S), **getc**(S), **putc**(S), **gets**(S), **puts**(S), **printf**(S), **scanf**(S)

FREXP(S)

Name

frexp, ldexp, modf - Splits floating-point number into a mantissa and an exponent.

Synopsis

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

Description

Frexp returns the mantissa of a double **value** as a double quantity, *x,* of magnitude less than 1, and stores an integer *n* such that **value** = *x *2** n* indirectly through *eptr.*

Ldexp returns the quantity **value***(2****exp**).

Modf returns the positive fractional part of **value** and stores the integer part indirectly through **iptr.**

FSEEK(S)

Name

fseek, ftell, rewind - Repositions a stream.

Synopsis

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

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 to the value of **ptrname**: 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).

See Also

lseek(S), **fopen(S)**

Diagnostics

Fseek returns nonzero for improper seeks, otherwise zero.

GAMMA(S)

Name

gamma - Performs log gamma function.

Synopsis

```
#include <math.h>
extern int signgam;

double gamma (x)
double x;
```

Description

Gamma returns **ln** | Γ (| x |) | . The sign of Γ (| x |) is returned in the external integer **signgam.** The following C program fragment might be used to calculate Γ:

```
y = gamma (x);
if (y > 88.0)
        error ();
y = exp (y) * signgam;
```

Diagnostics

For negative integer arguments, a huge value is returned, and *errno* is set to EDOM.

GETC(S)

Name

getc, getchar, fgetc, getw - Gets character or word from a stream.

Synopsis

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

Description

Getc returns the next character from the named input **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.** It returns the constant EOF upon end-of-file or error but because that is a valid integer value, **feof** and **ferror**(S) should be used to check the success of **getw. 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 on a read error.

Comments

Because **getc** is implemented as a macro, **stream** arguments with side effects are treated incorrectly. In particular, "getc(*f++)" doesn't work properly.

GETCWD(S)

Name

getcwd - Gets pathname of current working directory.

Synopsis

```
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) (See **malloc**(S).) If **pnbuf** is not a NULL pointer, 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.

See Also

pwd(CP), **malloc**(S), **popen**(S).

Diagnostics

Returns NULL with *errno* set if **maxlen** is not large enough, or if an error occurs in a lower-level function.

Comments

maxlen must be 2 more than the true length of the pathname.

GETENV(S)

Name

getenv - Gets value for environment name.

Synopsis

```
char *getenv (name)
char *name;
```

Description

Getenv searches the environment list (see **environ**(M)) for a string of the form *name* = *value* and returns *value* if such a string is present, otherwise 0 (NULL).

See Also

sh(C), **exec**(S)

GETGRENT(S)

Name

getgrent, getgrgid, getgrnam, setgrent, endgrent - Get group file entry.

Synopsis

```
#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.

> **Warning:** All information is contained in a static area, so it must be copied if it is to be saved.

GETLOGIN(S)

Name

getlogin - Gets login name.

Synopsis

```
char *getlogin ( )
```

Description

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used 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.

> **Warning:** The return values point to static data whose content is overwritten by each call.

GETOPT(S)

Name

getopt - Gets option letter from argument vector.

Synopsis

```
#include <stdio.h>

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

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 (that is, up to the first nonoption argument), **getopt** returns EOF. The special option **--** may be used to delimit the end of the options; EOF is returned, and **--** is skipped.

Diagnostics

Getopt prints an error message on **stderr** and returns a question mark **(?)** when it encounters an option letter not included in

optstring.

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 ();
    }
    for( ; optind < argc; optind++) {
            if (access (argv[optind], 4)) {
.
.
.
}
```

GETPASS(S)

Name

getpass - Reads a password.

Synopsis

```
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

> **Warning:** The return value points to static data whose content is overwritten by each call.

GETPID(S)

Name

getpid, getpgrp, getppid - Gets process, process group, and parent process IDs.

Synopsis

```
int getpid ()

int getpgrp ()

int getppid ()
```

Description

Getpid returns the process ID of the calling process. Most often **getpid** is used to generate unique temporary files.

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)

GETPW(S)

Name

getpw - Gets password for a given user ID.

Synopsis

```
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.

> **Warning:** This routine is shown only because it is included in some non-PC XENIX systems; it should not be used with PC XENIX. See **getpwent**(S) for routines to use instead.

GETPWENT(S)

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent - Gets password file entry.

Synopsis

```
#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** return 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.

> **Warning:** All information is contained in a static area so it must be copied if it is to be saved.

GETS(S)

Name

gets, fgets - Gets a string from a stream.

Synopsis

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

Description

Gets reads 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 the string is copied. **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.

Comments

Gets deletes the newline ending its input, but **fgets** keeps it.

GETUID(S)

Name

getuid, geteuid, getgid, getegid - Gets real user, effective user, real group, and effective group IDs.

Synopsis

```
int getuid ()

int geteuid ()

int getgid ()

int getegid ()
```

Description

The real user ID identifies the person who is logged in. This is in contra distinction to the effective user ID which determines the access permission at the moment.

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

introduction(S), **setuid**(S)

HYPOT(S)

Name

hypot, cabs - Determines Euclidean distance.

Synopsis

```
#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)

IOCTL(S)

Name

ioctl - Controls character devices.

Synopsis

```
#include <sys/ioctl.h>

ioctl (fildes, request, arg)
int fildes;
```

Description

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in Section M IBM Personal Computer *XENIX Command Reference* discuss how **ioctl** applies to them.

Ioctl fails if one or more of the following is true:

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 **tty**(M). [EINVAL]

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)

KILL(S)

Name

kill - Sends a signal to a process or a group of processes.

Synopsis

```
int kill (pid, sig)
int pid, sig;
```

Description

Kill sends a signal to a process or a group of processes 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 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 the 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 **introduction**(S)) and are referred to below as *proc0* and *proc1* respectively.

If **pid** is greater than zero, **sig** is sent to the process whose process ID is equal to **pid. Pid** may equal 1.

If **pid** is 0, **sig** is 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** is 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** is sent to all processes excluding *proc0* and *proc1*.

If **pid** is negative but not -1, **sig** is sent to all processes whose process group ID is equal to the absolute value of **pid.**

Kill fails and no signal is sent if one or more of the following is 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

On 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)

L3TOL(S)

Name

l3tol, ltol3 - Converts between three-byte integers and long integers.

Synopsis

```
l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

L3tol converts a list of **n** three-byte integers packed into a character string pointed to by **cp** into a list of long integers pointed to by **lp**.

Ltol3 performs the reverse conversion from long integers (**lp**) to three-byte integers (**cp**).

These functions are useful for file system maintenance where the block numbers are three bytes long.

See Also

filesystem(F)

LINK(S)

Name

link - Links a new filename to an existing file.

Synopsis

```
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 fails and no link is created if one or more of the following is 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's allocated address space. [EFAULT]

Return Value

On 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

ln(C), **unlink**(S)

LOCK(S)

Name

lock - Locks a process in primary memory.

Synopsis

```
lock (flag)
```

Description

If the **flag** argument is nonzero, the process executing this call is not swapped except if it is required to grow. If the argument is zero, the process is unlocked. This call may only be executed by the super-user.

Comments

Locked 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 processes soon after booting because that will tend to lock them into one end of memory.

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx. **Plock** provides the same function as **lock** and it is recommended for portability.

LOCKF(S)

Name

lockf - Provide semaphores and record locking on files.

Synopsis

```
#include <unlstd.h>

lockf (fildes, function, size)
int fildes, function;
long size;
```

Description

Lockf locks a specified region of the file given by the file descriptor, **fildes,** against access by all other processes. Other processes that attempt to use the locked region 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.

The *function* argument specifies what action to take. The possible values are:

F_ULOCK
Unlock a 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 EACCESS.

F_TEST
Test the region for other process 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.** If the **size** is 0, the region extends from the current position in the file to the current or future end of the file.

A process can create overlapping regions if desired. It cannot overlap regions locked by other processes. Adjacent regions locked by the same process are always combined into a single region.

A process can unlock all or part of any locked region. When regions are not fully unlocked, the remaining regions are still locked by the process. If the center of a locked region is unlocked, **lockf** creates two new locked regions from the remaining portions of the original region.

Return Values

On successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error. If a lock request is made and the table of active locks is full, *errno* is set to EDEADLK. If an attempt to unlock the center section of a locked region is made when no active lock table entries are available, *errno* is set to EDEADLK. If **fildes** is not a valid open file descriptor, *errno* is set to EBADF.

See Also

open(S), **creat**(S), **read**(S), **write**(S), **close**(S).

LOCKING(S)

Name

locking - Locks or unlocks a file region for reading or writing.

Synopsis

```
#include <sys/locking.h>

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 that attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked, depending on the mode in which the file region was locked. A process that attempts to write to or read from 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
Unlocks the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK
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
Locks the specified region. If any part of the region is already locked by a different process, returns the error EACCES instead of waiting for the region to become available for locking (nonblocking lockrequest).

LK_RLCK
Same as LK_LOCK except that the locked region may be read by other processes (read permitted lock).

LK_NBRLCK
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 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's locked area. Thus, calls to **locking, read,** or **write** scan for a deadlock before sleeping on a locked region. An 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 (that is: 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 is locked for read only, while the remaining region, if any, remains locked against reading and writing. There is no arbitrary limit to the number of regions that 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 that 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 is 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 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 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 EDEADLOCK. If there are no more free internal locks, *errno* is set to EDEADLOCK.

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx. **Lockf** provides the same function as **locking** and it is recommended for portability to other UNIX versions.

LOGNAME(S)

Name

logname - Finds login name of user.

Synopsis

```
char *logname()
```

Description

Logname returns a pointer to the null-terminated login name. It uses the string found in the LOGNAME variable from the user's environment.

Files

/etc/profile

See Also

env(C), **login**(M), **profile**(M), **environ**(M)

LSEARCH(S)

Name

lsearch - Performs linear search and update.

Synopsis

```
char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
int *nelp;
int 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 that are pointers to the elements being compared. The routine must return zero if the items are equal, and nonzero otherwise.

Comments

Unpredictable events can occur if there is not enough room in the table to add a new item.

See Also

bsearch(S), **qsort**(S)

LSEEK(S)

Name

lseek - Moves read/write file pointer.

Synopsis

```
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.**

On 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 remains unchanged if one or more of the following is 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

On 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)

MALLOC(S)

Name

malloc, free, realloc, calloc - Allocate main memory.

Synopsis

```
char *malloc (size)
 unsigned size;

free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

Description

Malloc and **free** provide a simple general-purpose memory allocation package. **Malloc** returns a pointer to a block of at least **size** bytes beginning on a word boundary.

The argument **free** is a pointer to a block previously allocated by **malloc;** this space is made available for further allocation, but its contents are left undisturbed.

Unpredictable results can occur if the 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 are unchanged up to the lesser of the new and old sizes.

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.

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.

MKNOD(S)

Name

mknod - Makes a directory or a special or ordinary file.

Synopsis

```
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:

```
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's effective user ID. The file's group ID is set to the process's effective group ID.

The low-order nine bits of **mode** are modified by the process's file mode creation mask; all bits set in the process's file mode creation mask are cleared. See **umask**(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, **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.

Mknod will fail and the new file will not be created if one or more of the following is true:

- The process's 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]
- 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's allocated address space. [EFAULT]

Return Value

On 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

mkdir(C), **mknod**(C), **chmod**(S), **creatsem**(S), **exec**(S), **sdget**(S), **umask**(S), **filesystem**(F)

Comments

Semaphore files should be created with the **creatsem**(S) system call.

Shared data files should be created with the **sdget**(S) system call.

MKTEMP(S)

Name

mktemp - Makes a unique filename.

Synopsis

```
char *mktemp (template)
char *template;
```

Description

Mktemp replaces **template** with a unique filename and returns a pointer to the name. The template should look like a filename with six trailing **X**'s, which will be replaced with the current process ID preceded by a zero.

See Also

getpid(S)

MONITOR(S)

Name

monitor - Prepares execution profile.

Synopsis

```
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 in the buffer a histogram of periodically sampled values of the program counter and of counts of calls of certain functions.

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)

Comments

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.

MOUNT(S)

Name

mount - Mounts a file system.

Synopsis

```
int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;
```

Description

Mount requests that a removable file system contained on the block special file identified by **spec** be mounted on the directory identified by **dir. Spec** and **dir** are pointers to pathnames.

On successful completion, references to the file **dir** refer to the root directory on the mounted file system.

The low-order bit of **rwflag** controls 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 fails if one or more of the following is 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's allocated address space. [EFAULT]

- **Dir** is currently mounted on someone's current working directory, or is otherwise busy. [EBUSY]

- The device associated with **spec** is currently mounted. [EBUSY]

Return Value

On 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)

NAP(S)

Name

nap - Suspends execution for a short interval.

Synopsis

```
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 received a signal while napping, the return value will be -1, and *errno* will be set to EINTR.

Comments

This function is driven by the system clock, which in most cases has a granularity of tens of milliseconds.

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

See Also

sleep(S)

NICE(S)

Name

nice - Changes priority of a process.

Synopsis

```
int nice (incr)
int incr;
```

Description

Nice adds the value of **incr** to the nice value of the calling process. A process's nice value is a positive number for which a 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 does not change the nice value if **incr** is negative and the effective user ID of the calling process is not super-user. [EPERM]

Return Value

On 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

nice(C), **exec**(S)

NLIST(S)

Name

nlist - Gets entries from name list.

Synopsis

```
#include <a.out.h>

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 returns -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.

OPEN(S)

Name

open - Opens file for reading or writing.

Synopsis

```
#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 ORing 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 returns without delay. An **open** for writing-only returns an error if no process currently has the file open for reading.

If O_NDELAY is clear:
An **open** for reading-only blocks until a process opens the file for writing. An **open** for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:
The open returns without waiting for carrier.

If O_NDELAY is clear:
The open blocks until carrier is present.

O_APPEND
If set, the file pointer is set to the end of the file before each write.

O_CREAT
If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's 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's 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** fails if the file exists.

O_SYNCW
Every write to this file descriptor is synchronous; that is, when the write system call ends, data is guaranteed to have been written to disk.

On 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 20 file descriptors open simultaneously.

The named file is opened unless one or more of the following is 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]
- Twenty 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's 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]

Return Value

On 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

close(S), **creat**(S), **dup**(S), **fcntl**(S), **lseek**(S), **read**(S), **write**(S)

Comments

The O–SYNCW flag is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions.

OPENSEM(S)

Name

opensem - Opens a semaphore.

Synopsis

```
sem_num = opensem (sem_name);
int sem_num;
char *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.

> **Warning:** It is not advisable to open the same semaphore more than once. Though it is possible to do this, it may result in a serious deadlock.

See Also

creatsem(S), **waitsem**(S), **sigsem**(S)

Diagnostics

Opensem returns the value -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 (that is, a file previously created by a process using a call to **creatsem),** *errno* is set to ENOTNAM. If the semaphore has become invalid because of inappropriate use, *errno* is set to ENAVAIL.

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

PAUSE(S)

Name

pause - Suspends a process until a signal occurs.

Synopsis

```
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** does 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)

PERROR(S)

Name

perror, sys__errlist, sys__nerr, errno - Send system error messages.

Synopsis

```
perror (s)
char *s;

externs int sys__nerr;
externs char *sys__errlist[ ];

externs int errno;
```

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

introduction(S)

PIPE(S)

Name

pipe - Creates an interprocess pipe.

Synopsis

```
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. The descriptors remain open across **fork**(S) system calls, making communication between parent and child possible.

Writes up to 5120 bytes of data 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 20 file descriptors open simultaneously.

Pipe fails if 19 or more file descriptors are currently open. [EMFILE]

Return Value

On 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)

PLOCK(S)

Name

plock - Lock process, text, or data in memory.

Synopsis

```
#include <sys/lock.h>

int plock (op)
int op;
```

Description

Plock allows the calling process to lock 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 the 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 is 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

On 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)

POPEN(S)

Name

popen, pclose - Initiates I/O to or from a process.

Synopsis

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

Description

The arguments to **popen** are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either "r" for reading or "w" for writing. **Popen** creates a pipe between the calling 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" command may be used 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.

Comments

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).

PRINTF(S)

Name

printf, fprintf, sprintf - Format output.

Synopsis

```
#include <stdio.h>

int printf (format[arg] . . . )
char *format;

int fprintf (stream, format[,arg] . . . )
FILE *stream;
char *format;

int sprintf (s, format[, arg ] . . . )
char *s, *format;
```

Description

Printf places output on the standard output stream **stdout. Fprintf** places output on the named output **stream. Sprintf** places output, followed by the null character (\0), in consecutive bytes starting at **s;** it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters 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 **args** under control of the **format.** The **format** is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more **args.** The results are undefined if there are insufficient **args** for the format. If the format is exhausted while **args** remain, the excess **args** are 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 is 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" (for example %04), the converted value is padded with zeroes. If the width is preceded with a blank (for example % 4), the value is 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 **args** specifying field width or precision must appear before the **arg** (if any) to be converted.

The flag characters and their meanings are:

-	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).

blank If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This implies that, if the blank and + flags both appear, the blank flag is ignored.

This flag specifies that the value is to be converted to an alternate form. For **c, d, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x (X)** conversion, a nonzero result has **0x (0X)** prepended to it. For **e, E, f, g,** and **G** conversions, the result always contains 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 are *not* removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer **arg** is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation **(x** and **X),** respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is 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, six digits are produced; if the precision is zero, no

decimal point appears. The **E** format code produces a number with **E** instead of **e** introducing the exponent. The exponent always contains exactly two digits.

g,G The float or double **arg** is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros 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)

PROFIL(S)

Name

profil - Creates an execution time profile.

Synopsis

```
profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;
```

Description

Buff points to an area of core whose length (in bytes) is given by **bufsiz.** After this call, the user's program counter is examined each clock tick, where a clock tick is some fraction of a second given in **machine**(M). **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.

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 noninterrupting 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 both child and parent after a **fork.** Profiling is turned off if an update in **buff** would cause a memory fault.

See Also

prof(CP), **monitor**(S)

PTRACE(S)

Name

ptrace - Traces a process.

Synopsis

```
#include <sys/types.h>

ptrace (request, pid, addr, data)
struct saddr *addr;
int request, pid, data;
```

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 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 on 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.

Sa__seg contains the selector for the Local Descriptor Table (LDT), the word at location *sa__off* in the LDT is returned. The **data** argument is ignored. These two requests 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. When executed in a segmented environment, *sa__seg* is ignored. The **data** argument is ignored. This request fails 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.

On successful completion, the value written into the address space of the child is returned to the parent. These two requests 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

7 This request causes the child to resume execution. If the **data** argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the **data** argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled.

The *sa_seg* of **addr** must be zero and the *sa_off* must be (int *)1. On successful completion, the value of **data** is returned to the parent. This request fails 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 behavior is somewhat CPU dependent.

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 stops 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 is 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]

A return value of -1 does not always indicate an error. To resolve this ambiguity, the *errno* variable is cleared on each call to **ptrace.** If the return value is -1, there is no error unless *errno* is nonzero.

Comments

The implementation and precise behavior 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.

System calls cannot be single-stepped. If a **ptrace** call requests a single step through a system call, the trace bit is cleared, and the user program runs to completion or until it encounters an explicitly set breakpoint.

See Also

adb(CP), **exec**(S), **signal**(S), **wait**(S), **machine**(M)

PUTC(S)

Name

putc, putchar, fputc, putw - Put a character or word on a stream.

Synopsis

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

Description

Putc appends the character **c** to the named output **stream.** 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 (that is, 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)) will cause it to become buffered; *setbuf,* again, will set 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

These functions return the constant EOF upon error. Because this is a valid integer, **ferror**(S) should be used to detect **putw** errors.

Comments

Because **putc** is implemented as a macro, the **stream** argument with side effects is not treated correctly.

PUTPWENT(S)

Name

putpwent - Writes a password file entry.

Synopsis

```
#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.

PUTS(S)

Name

puts, fputs - Puts a string on a stream.

Synopsis

```
#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)

Comments

Puts appends a newline, **fputs** does not.

QSORT(S)

Name

qsort - Performs a sort.

Synopsis

```
qsort (base, nel, width, compar)
char *base;
int 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 whether the first argument is to be considered less than, equal to, or greater than the second.

See Also

sort(C), **bsearch**(S), **lsearch**(S), **string**(S)

RAND(S)

Name

rand, srand - Generates a random number.

Synopsis

```
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.**

RDCHK(S)

Name

rdchk - Checks to see if there is data to be read.

Synopsis

```
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 (for example, EBADF), 0 if the process will block if it issues a *read*, and 1 if it is ok to read. EBADF is returned if a **rdchk** is done on a semaphore file or if the file specified doesn't exist.

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

READ(S)

Name

read - Reads from a file.

Synopsis

```
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, **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.

On 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 returns a 0.

If O_NDELAY is clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read returns a 0.

If O_NDELAY is clear, the read blocks until data becomes available.

Read fails if one or more of the following is true:

Fildes is not a valid file descriptor open for reading. [EBADF]

Buf points outside the allocated address space. [EFAULT]

Return Value

On successful completion, a nonnegative integer is returned, indicating the number of bytes actually read. Otherwise, a -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), **tty**(M)

Comments

Reading a region of a file locked with **locking** causes **read** to hang indefinitely until the locked region is unlocked.

REGEX(S)

Name

regex, regcmp - Compile and execute regular expressions.

Synopsis

```
char *regcmp (string1[,string2, . . . ],0)
char *string1, *string2, . . . ;

char *regex (re,subject[,ret0, . . . ])
char *re, *subject, *ret0, . . . ;
```

Description

Regcmp compiles a regular expression and returns a pointer to the compiled form. **Malloc**(S) is used to create space for the compiled expression. 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. Although **regcmp** and **regex** were derived from the editor, **ed**(C), 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 (for example, {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 is be stored in the **(n+1)**th argument following the subject argument. Ten enclosed regular expressions are allowed. **Regex** makes its assignments unconditionally.

(...)
Parentheses are used for grouping. An operator, for example *, +, {}, can work on a single character or a regular expression enclosed in parenthesis. For example: (a*(cb+)*)$0.

By necessity, all of 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 matches 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 matches through the string "Testing3" and returns the address of the character after the last matched character (cursor+11). The string "Testing3" is 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), **malloc**(S)

Comments

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;
}
```

REGEXP(S)

Name

regexp - Regular expression compile and match routines.

Synopsis

```
#define INIT<declarations>
#define GETC()<getc code>
#define PEEKC()<peekc code>
#define UNGETC( c ) <ungetc code>
#define Enter( 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), and others that perform regular expression-matching use this source file. Therefore, only this file needs to be changed to maintain regular expression compatibility.

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() return successive characters of the regular expression.

PEEKC() Return the next character in the regular expression. Successive calls to PEEKC() 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.

Enter(*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 that 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 that 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 1 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 that 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 is placed right after the declaration for the function **compile** and the opening brace ({). It is used for dependent declarations and initializations. It is also used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for **GETC(), PEEKC(),**and **UNGETC().** Otherwise, INIT can be used to declare external variables that might be used by **GETC(), PEEKC(),**and **UNGETC().** See the example of the declarations taken from **grep**(C).

There are other functions in this file that perform actual regular expression matching, one of which is the function **step.** The call to **step** 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 that was obtained by a call of the function **compile.**

The function **step** returns 1 if the given string matches the regular expression, and it returns 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 points to the first character of **string** and *loc2* points 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, **step** only tries 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* is saved for each compiled expression and *circf* is 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 1 indicating a match, or until the end of **string** is reached. If you want to constrain **string** to the beginning of the line in all cases, you do not need to call step; simply call **advance.**

When **advance** encounters a * or \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance** backs up along the string until it finds a match or reaches the point in the string that initially matched the \{ \} . 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** breaks out of the loop that backs up and returns 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 Enter(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).

Comments

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.

SBRK(S)

Name

sbrk, brk - Change data segment space allocation.

Synopsis

```
char *sbrk (incr)
int incr;

char *brk (addr)
char *addr;
```

Description

Sbrk and **brk** are used to dynamically change the amount of space allocated for the calling process's data segment; see **exec**(S). The change is made by resetting the process's break value. 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 large model programs, if **incr** is greater than the 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 reside entirely in one segment. If **incr** is negative and equal to the number of allocated bytes in the current data segment, the segment is automatically freed for other use. If **incr** is greater than the number of allocated bytes, the segment is freed, and the additional bytes are removed from the next data segment containing space allocated by **sbrk.**

Sbrk fails without making any change in the allocated space if such a change would result in more space being allocated than is allowed by a system-imposed maximum (see **ulimit**(S)). [ENOMEM]

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

On successful completion, **sbrk** and **brk** return pointers to the beginning of the allocated space. 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)

Comments

In 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.

SCANF(S)

Name

scanf, fscanf, sscanf - Convert and format input.

Synopsis

```
#include <stdio.h>

int scanf (format[, pointer ] . . . )
char *format;

int fscanf (stream, format[, pointer ] . . . )
FILE *stream;
char *format;

int sscanf (s, format[, pointer] . . . )
char *s, *format;
```

Description

Scanf reads from the standard input stream stdin. **Fscanf** reads from the named input **stream. Sscanf** reads from the character string **s.** Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string **format** described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. 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 *. 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 filled.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. 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.

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 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 a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a caret (^), the input field consists of all characters up to the first character that is not in the set between the brackets. If the first character after the left bracket is a ^, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d, o,** and **x** may be capitalized and/or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** and **f** may be capitalized and/or preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list.

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the last case, the offending 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**
.

The call:

```
int i; float x; char name[50];
scanf ("%2d%f%*d%2s", &i, &x, name);
```

with input:

56789 0123 45a72

will assign **56** to **i, 789.0** to **x,** skip **0123,** and place
the string **45\0**
in *name.*
The next call to **getchar** (see **getc**(S)) will return **a.**

See Also

atof(S), **getc**(S), **printf**(S)

Diagnostics

These functions return EOF on end of input and a short count for missing or illegal data items.

Comments

The success of literal matches and suppressed assignments is not directly determinable.

Trailing whitespace (including a newline) is left unread unless matched in the control string.

SDENTER(S)

Name

sdenter, sdleave - Synchronize access to a shared data segment.

Synopsis

```
#include <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. The actions performed depend on the value of **flags**. **Flags** values are formed by ORing 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, returns an 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.

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.

See Also

sdget(S), **sdgetv**(S)

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

SDGET(S)

Name

sdget,sdfree- Attach and detach a shared data segment.

Synopsis

```
#include <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 ORing 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, this flag has no effect. 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 is 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 an **sdenter** but not a **sdleave** for the specified segment, an **sdleave** is 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.**

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.

Comments

Use of the SD__UNLOCK flag on systems without hardware support for shared data may cause severe performance degradation.

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

See Also

sdenter(S), **sdgetv**(S)

SDGETV(S)

Name

sdgetv, sdwaitv - Synchronizes shared data access.

Synopsis

```
#include <sd.h>

int sdgetv (addr)
char *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

On 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)

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

SETBUF(S)

Name

setbuf - Assigns buffering to a stream.

Synopsis

```
#include <stdio.h>

setbuf (stream, buf)
FILE *stream;
char *buf;
```

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 tells how big an array is needed:

```
char buf[BUFSIZ];
```

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)

SETJMP(S)

Name

setjmp, longjmp - Performs a nonlocal "goto".

Synopsis

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

int 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 value 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 that calls **setjmp** must not itself have returned in the interim. **Longjmp** cannot return the value 0. If **longjmp** is invoked with a second argument of 0, it will return 1. All accessible data have values as of the time **longjmp** was called. The only exception to this are register variables. The value of register variables are undefined in the routine that called **setjmp** when the corresponding **longjmp** is invoked.

See Also

signal(S)

SETPGRP(S)

Name

setpgrp - Sets process group ID.

Synopsis

```
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.

Return Value

Setpgrp returns the value of the new process group ID.

See Also

exec(S), **fork**(S), **getpid**(S), **introduction**(S), **kill**(S), **signal**(S)

SETUID(S)

Name

setuid, setgid - Sets user and group IDs.

Synopsis

```
int setuid (uid)
int uid;

int setgid (gid)
int gid;
```

Description

Using **setuid** is comparable to having an *s* instead of an *x* in the execute field for the file owner. When this system call is run, it is given the permissions corresponding to the file owner. For example, the author of a game program can make the program **setuid** to the owner. This enables the owner to update a score file that is otherwise protected from other user's access.

The real user (group) ID of the current process is set to the argument **uid (gid)**. Both the effective ID and the real ID are set. The real user ID identifies the person that is logged in. This is in contra distinction to the effective user ID, which determines the access permission at this time. These calls are only permitted to the super user, unless the argument is the real ID or effective ID.

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.

Setuid (setgid) 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]

Return Value

On 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), **introduction**(S)

SHUTDN(S)

Name

shutdn - Flushes block I/O and halts the CPU.

Synopsis

```
#include <sys/param.h>
#include <sys/filsys.h>

shutdn (sblk)
struct filsys *sblk;
```

Description

Shutdn causes all information in core 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.

If **sblk** is nonzero, it specifies the address of a super-block which will be written to the root device as the last I/O before the halt. This facility is provided to allow file system repair programs to supersede the system's copy of the root super-block with one of their own.

Shutdn locks out all other processes while it is doing its work. However, it is recommended that user processes be ended (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)

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

SIGNAL(S)

Name

signal - Specifies what to do on receipt of a signal.

Synopsis

```
#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 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 of are described below.

The **SIG_DFL** value causes termination of the process on 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's open file descriptors are closed.

2. If the parent process of the receiving process is executing a **wait**, it is notified of the termination of the receiving process and the terminating signal's number is 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 is transformed into a zombie process (see **exit**(S) for definition of zombie process).

4. The parent process ID of each of the receiving process's existing child processes and zombie processes is set to 1. This means the initialization process (see **introduction**(S)) inherits each of these processes.

5. An accounting record is written on the accounting file if the system's accounting routine is enabled; see **acct**(S).

6. If the receiving process's process ID, tty group ID, and process group ID are equal, the signal **SIGHUP** is 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" is 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 filenamed **core** exists and is writable or can be created. If the file must be created, it has 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** is passed as the only argument to the signal-catching function. The consequences are:

1. Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted and the value of **func** for the caught signal is 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 because of the existence of a previously stopped or zombie process, the signal catching function is executed and the interrupted system call returns 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 fails if one or more of the following is true:

Sig is an illegal signal number, including SIGKILL. [EINVAL]

Func points to an illegal address. [EFAULT]

Return Value

On 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 XENIX; they are:

SIGCLD 18 Death of a child (not reset when caught)
SIGPWR 19 Power fail (not reset when caught)

These signals will continue to behave as described below; they are included only for compatibility with other versions of UNIX. Their use in new programs is strongly discouraged.

For these signals, **func** is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL- ignore signal
The signal is to be ignored.

SIG_IGN-ignore signal
The signal is to be ignored. Also, if **sig** is **SIGCLD**, the calling process's child processes do 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 are queued and the signal-catching function is 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** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the **func** value of **SIGCLD** is set to **SIG_IGN,** the exiting process does 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.

Comments

The defined constant NSIG in **signal.h** standing for the number of signals is always at least one greater than the actual number.

SIGSEM(S)

Name

sigsem - Signals a process waiting on a semaphore.

Synopsis

```
sigsem (sem_num)
int sem_num;
```

Description

Sigsem signals a process that is waiting for 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 are put to sleep until the process that currently has access to the resource issues a **sigsem** call. If there are any waiting processes, **sigsem** causes the process that 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)

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 by **opensem,** *errno* is set to EBADF. If the process issuing a sigsem call is not the current "owner" of the semaphore (that is, if the process has not issued a waitsem call before the sigsem), *errno* is set to ENAVAIL.

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

SINH(S)

Name

sinh, cosh, tanh - Perform hyperbolic functions.

Synopsis

```
#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 a huge value of appropriate sign when the correct value would overflow.

SLEEP(S)

Name

sleep - Suspends execution for an interval.

Synopsis

```
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 terminates the **sleep** following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system. The value returned by **sleep** is 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 because of 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)

SSIGNAL(S)

Name

ssignal, gsignal - Implement software signals.

Synopsis

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

Description

Ssignal and **gsignal** implement a software facility similar to **signal**(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,** 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.

Comments

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.

STAT(S)

Name

stat, fstat - Get file status.

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

Description

Path points to a 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** is defined in the **<sys/stat.h>** include file (see **stat**(F)).

Several XENIX functions cause the status information of a given file to be updated. The last access time (*st_atime*) for a file is updated by the following system calls: **creat**(S), **mknod**(S), **pipe**(S), **utime**(S), and **read**(S). The last modification time (*st_mtime*) is updated by: **creat**(S), **mknod**(S), **pipe**(S), **utime**(S),

and **write**(S). The last change of status *st__ctime* for a file is updated by: **chmod**(S), **chown**(S), **creat**(S), **link**(S), **mknod**(S), **pipe**(S), **utime**(S), and **write**(S).

The device identification value (*st__rdev*) contains the device major and minor numbers for the given file only if that file is a character or block special file. If the file is a shared memory or semaphore file, it contains the type code. Note that the file */usr/include/sys/types.h* contains the macros *major*() and *minor*() for extracting major and minor numbers from *st__rdev*. See **stat**(F) for the semaphore and shared memory type code values S__INSEM and S__INSHD.

Stat fails if one or more of the following is 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 fails if one or more of the following is true:

- **Fildes** is not a valid open file descriptor. [EBADF]
- **Buf** points to an invalid address. [EFAULT]

Return Value

On 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)

STDIO(S)

Name

stdio - Performs standard buffered input and output.

Synopsis

```
#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 disrupt the program, possibly including program termination. Individual function descriptions describe the possible error conditions.

STIME(S)

Name

stime - Sets the time.

Synopsis

```
#include <sys/types.h>
#include <sys/timeb.h>

time_t 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

On 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)

STRING(S)

Name

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strdup - Perform string operations.

Synopsis

```
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, c;

char *strrchr (s, c)
char *s, 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 as **s1** is lexicographically greater than, equal to, or less than **s2. Strncmp** makes the same comparison but looks at most **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 nonnull 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 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.

Comments

Strcmp uses native character comparison, which is signed on some machines, unsigned on others.

All string movement is performed character-by-character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may yield surprises.

SWAB(S)

Name

swab - Swaps bytes.

Synopsis

```
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.

SYNC(S)

Name

sync - Updates the super-block.

Synopsis

```
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 that examine a file system, for example **fsck**(C), **df**(C), etc.

The writing, although scheduled, is not necessarily complete on return from **sync.**

See Also

sync(C)

SYSTEM(S)

Name

system - Executes a shell command.

Synopsis

```
#include <stdio.h>

int system (string)
char *string;
```

Description

System passes the **string** to a new invocation of a shell (see **sh**(C)). The shell reads and executes the *string* as if it had been typed as a command at a terminal, then returns the exit status of the command to the calling process. The calling process waits until the shell has returned a status before proceeding with execution.

See Also

sh(C), **exec**(S)

Diagnostics

System stops if it can't execute **sh** (C).

TERMCAP(S)

Name

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - Performs terminal functions.

Synopsis

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent (bp,name)
char *bp, *name;

tgetnum (id)
char *id;

tgetflag (id)
char *id;

char *tgetstr (id, area)
char *id, **area;

char *tgoto (cm, destcol, destline)
char *cm;

tputs (cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

Description

These functions extract and use capabilities from the terminal capability data base **termcap**(M). 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 will look 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 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**(M), 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 that call **tgoto** should be sure to turn off the TAB3 bit (see **tty**(M)), because **tgoto** may now output a tab. Note that programs using termcap should in general turn off TAB3 anyway because some terminals use Ctrl-I for other functions, such as nondestructive space.) If a % sequence is given that is not understood, **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 that is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by **stty**(C). 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.

Comments

These routines can be linked by using the linker option - ltermcap.

TIME(S)

Name

time, ftime - Get time and date.

Synopsis

```
long time ((long*) 0)

long time (tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>

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 "Return Value" below .)

Time fails if **tloc** points to an illegal address. [EFAULT]
Likewise, **ftime** fails if **tp** points to an illegal address. [EFAULT]

Return Value

On 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 {
        time__t 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), **time**(S), **ctime**(S)

Comments

Since **ftime** does not return the correct timezone value, its use is not recommended. See **ctime**(S) for accurate use of the TZ variable.

Ftime is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. **Ftime** must be linked with the compiler/linker option -lx.

TIMES(S)

Name

times - Gets process and child process times.

Synopsis

```
#include <sys/types.h>
#include <sys/times.h>

long times(buffer)
struct tms *buffer;
```

Description

Times fills the structure pointed to by **buffer** with time-accounting information. The contents of the structure is:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

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 *tms_utimes* and *tms_cutimes* of the child processes.

Tms_cstime is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

Times fails if **buffer** points to an illegal address. [EFAULT]

Return Value

On 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)

TMPFILE(S)

Name

tmpfile - Creates a temporary file.

Synopsis

```
#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.

See Also

creat(S), **unlink**(S), **fopen**(S), **mktemp**(S), **tmpnam**(S)

TMPNAM(S)

Name

tmpnam - Creates a name for a temporary file.

Synopsis

```
#include <stdio.h>

char *tmpnam (s)
char *s;
```

Description

Tmpnam generates a filename that can safely be used for a temporary file. If (int) s is zero, **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 (int) s is nonzero, **s** is assumed to be the address of an array of at least **L_tmpnam** bytes; **tmpnam** places its result in that array and returns **s** as its value.

Tmpnam generates a different filename each time it is called.

Files created using **tmpnam** and either **fopen** or **creat** are only temporary 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.

See Also

creat(S), **unlink**(S), **fopen**(S), **mktemp**(S)

Comments

If called more than 17,576 times in a single process, **tmpnam** starts 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 **tmpnam** or **mktemp** and the filenames are chosen so as to render duplication by other means unlikely.

TRIG(S)

Name

sin, cos, tan, asin, acos, atan, atan2 - Perform trigonometric functions.

Synopsis

```
#include <math.h>

double sin (x)
double x;

double cos (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.

Diagnostics

Arguments of magnitude greater than 1 cause **asin** and **acos** to return value 0.

Comments

These routines can be linked with the linker option **-lm**.

TTYNAME(S)

Name

ttyname, isatty - Finds the name of a terminal.

Synopsis

```
char *ttyname (fildes)
int 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**.

Comments

The return value points to static data whose contents are overwritten by each call.

ULIMIT(S)

Name

ulimit - Gets and sets user limits.

Synopsis

```
#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
Gets the process's 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
Sets the process's 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. **Ulimit** fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

UL_GMEMLIM
Gets the maximum possible break value. If the process is a large model 80286 program, 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

On 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

sbrk(S), **chsize**(S), **write**(S)

Comments

The file limit is only enforced on writes to regular files. Tapes, disks, and other devices of any size can be written.

UMASK(S)

Name

umask - Sets and gets file creation mask.

Synopsis

```
int umask (cmask)
int cmask;
```

Description

Umask sets the process's file mode creation mask to **cmask** and returns the previous value of the mask. Only the low-order nine 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)

UMOUNT(S)

Name

umount - Unmounts a file system.

Synopsis

```
int umount (spec)
char *spec;
```

Description

Umount requests that a previously mounted file system contained on the block special device identified by **spec** be unmounted. **Spec** is a pointer to a 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 fails if one or more of the following is true:

- The process's effective user ID is not super-user. [EPERM]
- **Spec** does not exist. [ENXIO]
- **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's allocated address space. [EFAULT]

Return Value

On 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)

UNAME(S)

Name

uname - Gets name of current XENIX system.

Synopsis

```
#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];
        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. *Release* and *version* further identify the operating system. *Sysorigin* and *sysoem* identify the source of the XENIX version. *Sysserial* is a software serial number that may be zero if unused.

Uname fails if **name** points to an invalid address. [EFAULT]

Return Value

On successful completion, a nonnegative value is returned. Otherwise, -1 is returned, and *errno* is set to indicate the error.

See Also

uname(C)

Comments

Not all fields may be set on a particular system.

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions.

UNGETC(S)

Name

ungetc - Pushes character back into input stream.

Synopsis

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

Description

Ungetc pushes the character **c** back on an input stream. The character is 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.

UNLINK(S)

Name

unlink - Removes directory entry.

Synopsis

```
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 is 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. [EACCES]
- 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's 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

On 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)

USTAT(S)

Name

ustat - Gets file system statistics.

Synopsis

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

Description

Ustat returns information about a mounted file system. **Dev** is the major/minor (1 byte each) device number identifying a device containing a mounted file system. (See the makedev, major, and minor macros in the types.h file for more details). **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 */
```

One example of this call is:

```
ustat(makedev(major(x),minor(y)),buf);
```

where x and y are 2-byte integers representing the major and minor device numbers.

Ustat fails if one or more of the following is true:

> **Dev** is not the device number of a device containing a mounted file system. [EINVAL]

Buf points outside the process's allocated address space. [EFAULT]

Return Value

On 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)

Comments

When using file systems from previous versions of UNIX, **fsck**(C) must be run on the file system before mounting. Otherwise the **ustat** system call does not work correctly. This only needs to be done once.

UTIME(S)

Name

utime - Sets file access and modification times.

Synopsis

```
#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 fails if one or more of the following is 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's allocated address space. [EFAULT]
- **Path** points outside the process's allocated address space. [EFAULT]

Return Value

On 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)

WAIT(S)

Name

wait - Waits for a child process to stop or terminate.

Synopsis

```
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's child processes stops in a trace mode (see **ptrace**(S)) or terminates. If a child process stopped or terminated before 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 as following:

If the child process stopped, the high-order 8 bits of status are zero, and the low-order 8 bits are set equal to 0177.

If the child process terminated because of an **exit** call, the low-order 8 bits of status are zero and the high-order 8 bits contain the low-order 8 bits of the argument that the child process passed to **exit ;** see **exit**(S).

If the child process terminated because of a signal, the high-order 8 bits of status are zero, and the low-order 8 bits

contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (that is, 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 **introduction**(S).

Wait fails and returns immediately if one or more of the following is 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 because of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If **wait** returns because of 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).

WAITSEM(S)

Name

waitsem, nbwaitsem - Await and check access to a resource governed by a semaphore.

Synopsis

```
waitsem (sem_num)
int sem_num;

nbwaitsem (sem_num)
int sem_num;
```

Description

Waitsem gives the calling process access to the resource governed by the semaphore **sem_num.** If the resource is in use by another process, **waitsem** puts the process to sleep until the resource becomes available; **nbwaitsem** returns the error ENAVAIL. **Waitsem** and **nbwaitsem** are used with **sigsem** to allow synchronization of processes wishing to access a resource. One or more processes may **waitsem** on the given semaphore and are put to sleep until the process that currently has access to the resource issues **sigsem. Sigsem** causes the process that 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), **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) for the semaphore when the process controlling the semaphore exits without relinquishing control (thereby leaving the resource in an undeterminate state), return with *errno* set to ENAVAIL. If a process does two **waitsems** in a row without doing an intervening **sigsem**, *errno* is set to EINVAL.

Comments

This feature is an IBM Personal Computer XENIX improvement and may not be present in all UNIX versions. The application developer should consider the impact to portability when using this feature. This routine must be linked with the compiler/linker option -lx.

WRITE(S)

Name

write - Writes to a file.

Synopsis

```
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 before each write.

Write fails and the file pointer remains unchanged if one or more of the following 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's file size limit or the maximum file size. See **ulimit**(S). [EFBIG]

Buf points outside the process's allocated address space. [EFAULT]

If a **write** requests that more bytes be written than there is room for (for example, the uimit (see **ulimit**(S)) or the physical end of a medium), only as many bytes as there is room for are 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 fails if a write of **nbyte** bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O__NDELAY flag of the file flag word is set, 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

On 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)

Comments

Writing a region of a file locked with **locking** causes **write** to hang indefinitely until the locked region is unlocked.

Appendix A. System Call and Library Function Cross Reference

This section lists the functions found in various libraries, and the functions that directly invoke system primitives.

System Calls

access	fstat	mount	stat
acct	getpid	nice	stime
alarm	getpgrp	open	sync
brk	getppid	pause	time
chdir	getuid	pipe	times
chmod	geteuid	plock	ulimit
chown	getgid	profil	umask
chroot	getegid	ptrace	umount
close	ioctl	read	uname
creat	kill	sbrk	unlink
dup	link	setgid	ustat
exit	lockf	setpgrp	utime
fcntl	lseek	setuid	wait
fork	mknod	signal	write

Extended System Calls

chsize	nap	sdgetv	waitsem
creatsen	nbwaitsem	sdenter	
dup2	opensem	sdleave	
ftime	rdchk	sdwaitv	
lock	sdget	shutdn	
locking	sdfree	sigsem	

Note: Extended System Calls are included at link time by specifying -lx to the compiler or linker.

Library Routines

The following libraries are provided as standard with Xenix. On 8086/88 and 286 systems, versions for Small, Middle, and Large model programs are provided (for example, three of each library).

They are included at link time by specifying -l*name* to the compiler or linker, where *name* is the name listed below less the *lib* prefix. For example -*lm*, and -*ltermcap*.

libc	The standard library containing all system call interfaces, Standard I/O routines, and other general purpose services. Libc is the default library and need not be explicitly specified.
libx	Extended system calls that may not be present in other UNIX implementations.
libm	The standard math library.
libl	Library for use with programs produced by lex.
liby	Library for use with programs produced by yacc.
libtermcap	Routines for accessing the termcap data base describing terminal characteristics.
libtermlib	The same as libtermcap.
libcurses	Screen and cursor manipulation routines
libdbm	Data base management routines

The Standard C Library - libc

a64l
abort
abs
asctime
assert
atof
atoi
atol
bsearch
calloc
clearerr
ctermid
ctime
cuserid
defopen
defread
ecvt
endgrent
endpwent
fclose
fcvt
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
fopen
fprintf
fputc
fputs
fread
free
freopen
frexp
fscanf
fseek
ftell
fwrite
fxlist
gcvt
getc
getchar
getenv
getgrent
getgrgid
getgrnam
getlogin
getopt
getpass
getpw
getpwent
getwnam
getpwuid
gets
getw
gmtime
gsignal
isalnum
isalpha
isascii
isatty
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
l3tol
l64a
ldexp
localtime
logname
longjmp
lsearch
ltol3
malloc
mktemp
modf
monitor
nlist
pclose
perror
popen
printf
putc
putchar
putpwent
puts
putw
qsort
rand
realloc
regcmp
regex
rewind
scanf
setbuf
setgrent
setjmp
setpwent
sleep
sprintf
srand
sscanf
ssignal
strcat
strchr
strcmp
strcpy
strcspn
strdup
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strtok
swab
system
tmpfile
tmpnam
toascii
tolower
toupper
ttyname
tzset
ungetc
xlist

The Standard Math Library - libm

acos
asin
atan
atan2
cabs
ceil
cos
cosh
exp
fabs
floor
fmod
gamma
hypot
j0
j1
jn
log
log10
pow
sin
sinh
sqrt
tan
tanh
y0
y1
yn

The Default Lex Library - libl

main
yyless
yywrap

The Default Yacc Library - liby

main
yyerror

The Terminal Capabilities Library - libtermcap

tgetent
tgetflag
tgetnum
tgetstr
tgoto
tputs

The Screen Manipulation Library - libcurses

curses

The Data Base Management Library - libdbm

dbminit
delete
fetch
firstkey
nextkey
store

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

The Personal Computer
Programming Family

Reader's Comment Form

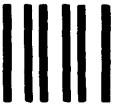
XENIX™ 6138822
Software Command Reference

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

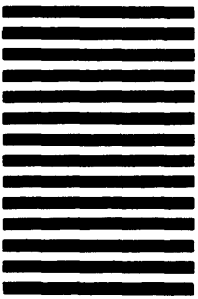
IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

International Business
Machines Corporation
P.O. Box 1328-S
Boca Raton, Florida 33432

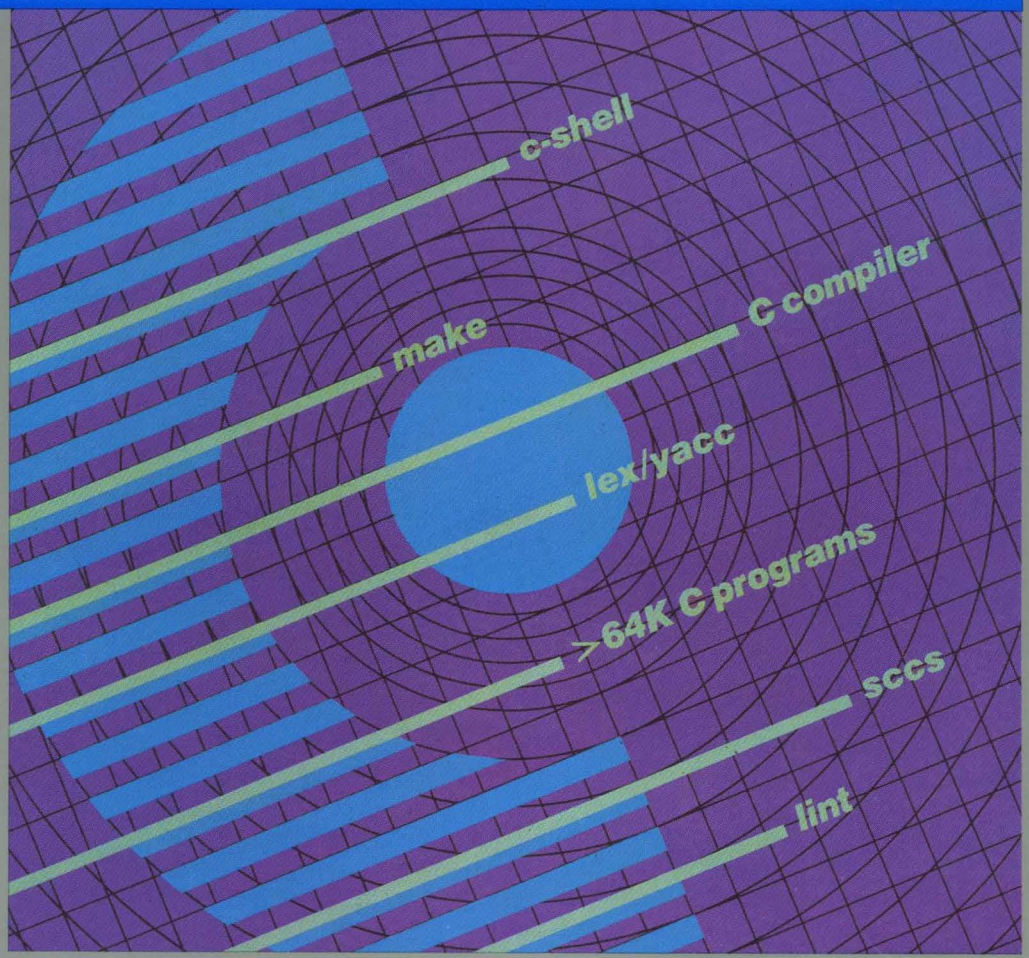
Printed in the
United States of America

6138822

Software Command Reference

IBM Personal Computer XENIX™ Software Development System

Programming Family



IBM
Personal Computer Software

Software development tools, including language translators, source code management tools, a C compiler, a debug facility and a linker for combining modules into finished programs. The C compiler generates code for DOS or the IBM Personal Computer XENIX™ Operating System.

Software required:

IBM Personal Computer XENIX™ Operating System

Software included:

Three 1.2MB diskettes

System requirements:

IBM Monochrome or Color Display or equivalent (with appropriate adapter)

IBM Personal Computer AT™

512KB RAM memory

IBM 20MB fixed disk

IBM 1.2MB diskette drive

Note:
XENIX is a trademark of Microsoft Corporation.

International Business Machines Corporation
P.O. Box 1328-S
Boca Raton, Florida 33432

Printed in the United States of America

6024209

International Business Machines Corporation **Boca Raton, Florida 33432**
IBM Program License Agreement

You should carefully read the following terms and conditions before opening this diskette package. Opening this diskette package indicates your acceptance of these terms and conditions. If you do not agree with them, you should promptly return the package unopened; and your money will be refunded.

IBM provides this program and licenses its use in the United States and Puerto Rico. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

License

You may:

a. use the program on a single machine;

b. copy the program into any machine readable or printed form for backup or modification purposes in support of your use of the program on the single machine (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected.");

c. modify the program and/or merge it into another program for your use on the single machine (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement.); and,

d. transfer the program and license to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the program contained or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification or portion merged into another program.

You may not use, copy, modify, or transfer the program, or any copy, modification or merged portion, in whole or in part, except as expressly provided for in this license.

If you transfer possession of any copy, modification or merged portion of the program to another party, your license is automatically terminated.

Term

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

Limited Warranty

The program is provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you (and not IBM or an authorized Personal Computer dealer) assume the entire cost of all necessary servicing, repair or correction.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassettes on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

Limitations of Remedies

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette or cassette not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized **IBM Personal Computer** dealer with a copy of your receipt.

or

2. if IBM or the dealer is unable to deliver a replacement diskette or cassette which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damages arising out of the use or inability to use such program even if IBM or an authorized IBM Personal Computer dealer has been advised of the possibility of such damages, or for any claim by any other party.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

General

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W Boca Raton, Florida 33432.

You acknowledge that you have read this agreement, understand it and agree to be bound by its terms and conditions. You further agree that it is the complete and exclusive statement of the agreement between us which supercedes any proposal or prior agreement, oral or written, and any other communications between us relating to the subject matter of this agreement.