K PROGRAMMERS REFERENCE

Part No. 061-3207-00
Product Group 07

# 4400 SERIES C LANGUAGE

Tektronix
COMMITTED TO EXCELLENCE

TEK PROGRAMMERS REFERENCE

Part No. 061-3207-00
Product Group 07

# 4400 SERIES C LANGUAGE

*Please Check at the Rear of this Manual for NOTES and CHANGE INFORMATION*

First Printing MAR 1986

Tektronix®
COMMITTED TO EXCELLENCE

## WARRANTY FOR SOFTWARE PRODUCTS

**PLEASE FORWARD ALL MAIL TO:**

**Artificial Intelligence Machines**
**Tektronix, Inc.**
**P.O. Box 1000 M.S. 60-405**
**Wilsonville, Oregon 97070**
**Attn: AIM Documentation**

# MANUAL REVISION STATUS

**PRODUCT: 4400 SERIES C LANGUAGE PROGRAMMERS REFERENCE**

This manual supports the following versions of this product: **4404 Version 1.5, 4405 Version 1.1, and 4406 Version 1.1.**

| REV DATE | DESCRIPTION |
|---|---|
| MAR 1986 | Original Issue |

# Section Table of Contents

SECTION 1 Introduction

About This Manual ........ 1-1
Where to find Information ........ 1-1
Manual Syntax Conventions ........ 1-2
Invoking the C Compiler ........ 1-2
The Command Line ........ 1-3
- The a Option ........ 1-5
- The c Option ........ 1-5
- The D Option ........ 1-5
- The f Option ........ 1-5
- The i Option ........ 1-6
- The I Option ........ 1-6
- The l Option ........ 1-6
- The L Option ........ 1-7
- The m Option ........ 1-7
- The M Option ........ 1-7
- The n Option ........ 1-7
- The N Option ........ 1-7
- The o Option ........ 1-7
- The O Option ........ 1-8
- The q Option ........ 1-8
- The r Option ........ 1-9
- The R option ........ 1-9
- The t Option ........ 1-9
- The U Option ........ 1-9
- The v Option ........ 1-9
- The w Option ........ 1-10
- The x Option ........ 1-10

Examples ........ 1-11
Description of the Language ........ 1-12
- Object Sizes ........ 1-12
- Register Variables ........ 1-12

SECTION 2 Kernighan and Ritchie Variations

Introduction ........ 2-1
Identifiers (Names), page 179 ........ 2-1
- Character Constants, page 180 ........ 2-1
- Enumeration Constants, new section ........ 2-1

Hardware Characteristics, page 181 ........ 2-1
- What's in a Name?, page 182 ........ 2-2

Characters and Integers, page 183 ........ 2-2
Void, new section ........ 2-2
Type Specifiers, page 193 ........ 2-2
Structure, Union, and Enumeration Declarations, page 196 ........ 2-3
Inclusion of an Information Field, new section ........ 2-4
Structures and Unions, page 209 ........ 2-4
Explicit Pointer Conversions, page 210 ........ 2-4
Portability Considerations, page 211 ........ 2-4
Anachronisms, page 212 ........ 2-4

SECTION 3 System Calls and Functions

SECTION 4 Graphics Library Concepts

The Graphics Library ..... 4-1
About This Section ..... 4-1
Using the Graphics Library in C Programs ..... 4-1
Using the Graphics Library in Assembly Language Programs ..... 4-2
Graphics Environment and Structures ..... 4-3
Entering and Exiting Graphics Mode ..... 4-3
Environmental Settings ..... 4-4
Saving and Restoring the Display State ..... 4-4
DISPSTATE ..... 4-4
Panning ..... 4-5
Mouse Bounds ..... 4-5
Viewport ..... 4-5
Cursors and Halftones ..... 4-5
Screen Size Constants ..... 4-6
Graphics Structures ..... 4-6
POINT ..... 4-6
RECT ..... 4-6
FORM ..... 4-7
FONT ..... 4-8
MENU ..... 4-8
Bitblt Graphics ..... 4-8
The BBCOM Structure ..... 4-9
How the BitBlt Operation Works ..... 4-12
Drawing a Box on the Screen ..... 4-14
Initializing Graphics Mode ..... 4-15
Setting Up the BitBlt Structure ..... 4-15
Drawing the Box ..... 4-16
An Interesting Example Program ..... 4-16
Graphics Error Messages ..... 4-19
Creating Images In Forms ..... 4-19
System Fonts ..... 4-20
Font Styles and Layout ..... 4-20
Graphical Text ..... 4-24
Menus ..... 4-25
Pop-Up Menus ..... 4-25
Text Menus ..... 4-25
Icon Menus ..... 4-27
Event Processes ..... 4-28

SECTION 5 Graphics Library Reference

Graphics and Events Library ..... 5-1
About This Section ..... 5-1
List of Functions ..... 5-2

## Figures

4-1. A 16 by 16 Bit Form. ..... 4-7

4-2. The BBCOM Data Structure. .................... 4-9
4-3. The BitBlt Operation. .................... 4-12
4-4. Tektronix Proportional Fonts (PellucidaSerif and PellucidaSans-Serif). .................... 4-21
4-5. Tektronix Monospaced Fonts (Pellucida Typewriter) Part 1. .................... 4-22
4-6. Tektronix Monospaced Fonts (Pellucida Typewriter) Part 2. .................... 4-23
4-7. Event Queue Processing. .................... 4-28

## Tables

1-1 Variable Sizes .................... 1-12
4-1 Statebits definitions .................... 4-5
5-1 Symbolic Arguments .................... 5-2

# Section 1

# Introduction

## About This Manual

This manual is the primary programmer's reference to the 4400 Series C language. This manual contains manual pages for C language functions and system calls as well as graphics library functions. The *4400 Users Manual* contains a complete list of the other manuals available for the 4400 Series.

This manual has these sections:

**Section 1 Introduction.** Tells you about this manual and also tells you how to invoke the C compiler, and the options that are available for the command string.

**Section 2 Kernigan and Ritchie Variations.** Provides you with information about how the 4400 Series C language implementation is different from the implementation described in Kernigan and Ritchie's *The C Programming Language.*

**Section 3 Functions and System Calls.** A description of the C language function and system calls available on the 4400 Series.

**Section 4 Graphics Library Concepts.** An introduction to graphics, fonts, and event processes on the 4400 Series AIM systems. BitBlt graphics concepts are discussed in some detail.

**Section 5 Graphics Library Reference.** A description of the C and assembly language callable graphics library functions on the 4400 Series systems.

## Where to find Information

You have several important sources of information on the 4400:

- This manual, the *4400 Series C Language Reference* manual, contains reference manual pages for C language function and system calls as well as graphics library functions.
- The *4400 Series Operating System Reference* manual contains the syntax and details of commands and utilities. This manual also contains details about a text editor and a remote terminal emulator.
- The *4400 Series Assembly Language Programmers Reference* manual contains the details of the assembler and linking loader.
- The *4400 Users* manual contains basic information on system installation, startup, installing software, and the other "how to put commands together" discussions. See the index of the *User's* manual to find how to perform particular tasks.
- The on-line *help* utility contains a brief description of the syntax of user commands.
- The *Introduction to Smalltalk-80(tm)* manual contains details and a short tutorial on the Smalltalk-80 programming language.

- The reference manuals for the optional languages for the 4400 product family are also availabe.

## Manual Syntax Conventions

Throughout this manual, the *4400 User's* manual, and in the on-line help files, the following syntax conventions apply:

1. Words standing alone on the command line are *keywords*. They are the words recognized by the system and should be typed exactly as shown.
2. Words enclosed by angle brackets (< and >) enclose descriptions that are replaced with a specific argument. If an expression is enclosed *only* in angle brackets, it is an essential part of the command line. For example, in the line:

   ```
   adduser <user_name>
   ```

   you must specify the name of the user in place of the expression <user_name>.
3. Words or expressions surrounded by square brackets ([ and ]) are optional. You may omit these words or expressions if you wish.
4. If the word *list* appears as part of a term, that term consists of one or more elements of the type described in the term, separated by spaces. For example:

   ```
   <file_name_list>
   ```

   consists of a series (one or more) of file names separated by spaces.

## Invoking the C Compiler

The **cc** command is the program that drives the C compiler. By default, **cc** calls the two passes of the C compiler — the relocating assembler, and the linking-loader. In addition, if you specify the **O** option, **cc** calls the assembly language optimizer. Numerous options let you pass information to the programs called by **cc** and to control their execution. The **cc** command produces code that does not check the availability of stack space before trying to obtain space on the stack.

The driver program accepts as input C source files, relocatable modules, or both. When you specify either the **r** or **R** option, you may also use assembly language files as input. The name of a C source file must end in **.c**; of an assembly language file, in **.a**; of a relocatable module, in **.r**; of a preprocessor file in **.p**.

By default, the **cc** command produces an output file named accordingly:

- If the user specifies only one file on the command line and that file is a file named *<filename>*.**c** containing C source code, the output file is named *<filename>*.
- Otherwise, the output file is named *<output>*.

You may override this naming procedure by using the **o** option to **cc** to specify the name of the output file. In any case, if a file with the same name already exists, it is deleted with no warning.

The **cc** command can produce as output one or more files containing intermediate language (from the first pass of the C compiler), assembly language (from the second pass of the C compiler), relocatable binary code, or executable binary code. You can obtain a listing of the C source code by specifying the **L** or **N** option to **cc**. This listing is written to standard output.

Whether or not the **cc** command produces code that checks the availability of stack space before trying to obtain space on the stack depends on the type of system being used. If it does, each time the program needs space on the stack, it calls a run-time routine, which ensures that space is available by adding to the stack if necessary. The **cc** command automatically produces the correct code for a given system, but the user may override the default for a particular system by using the s or S option. Code that does not check the availability of stack space is both smaller and faster than code that does.

Compilation errors are always sent to standard output with the offending line of code and the line number.

## The Command Line

The syntax for invoking the C compiler is:

cc *<file_name_list> [+acDfiIlLmMnNoOqrRtUvwx]*

where *<file_name_list>* is a list of the names of the files to compile, assemble, and link. The items in brackets are options that can be used in the command line. Brief descriptions of the options that are available are given here. These options are discussed in more detail later in this section.

| | |
|---|---|
| a | Stop when the second pass of the C compiler is complete. |
| c | Put the comments generated by the C compiler into the assembly language file. |
| D<*symbol*> [=*def*] | Define the specified symbol. |
| f | Produce an output module suitable for firmware. |

| | |
|---|---|
| i=<*dir_name*> | Specifies a directory to search for *#include* files. |
| I | Stop when the first pass of the C compiler is complete. |
| l=<lib_name> | Specifies the name of a library to pass to the linking-loader. |
| L | Send to standard output a listing of those files containing C source code. Expand *#include* files. |
| m | Tell the linking-loader to produce load and module maps. |
| M | Tell the linking-loader to produce as output one relocatable file. The name of this file is **output.r**. |
| n | Call only the first pass of the C compiler. Do not produce any code. |
| N | Send to standard output a listing of those files containing C source code. Do not expand *#include* files. |
| o=<*filename*> | Specifies the name of the executable (or if the **M** option is in effect, the relocatable) output file. |
| O | Call the assembly language optimizer. |
| q | Produce code that does calculations on *char* and *short* variables without first converting to *int*. |
| r | Tell the assembler to produce a relocatable module from each input file, but do not call the linking-loader. The **r** option leaves the user with one relocatable module for each input file. |
| R | Tell the assembler to produce a relocatable module from each input file. Then call the linking-loader, but do not delete the relocatable modules. |
| t | Produce as output a shared-text, executable module. |
| U | Produce a line-feed character ($0A) for *\n* rather than the default of a carriage return ($0D). |
| v | Use verbose mode. When this option is in effect, the **cc** command sends messages to standard error describing its activities. |
| w | Warn about duplicate *#define* statements. |
| x=<*ldr_option*> | Pass the information following the equal sign to the linking-loader. With this option the user can pass any option to the linking-loader. |

Detailed descriptions of these options follow.

## The a Option

The **a** option instructs the **cc** command to stop when the second pass of the C compiler is complete. The name of each output file is the same as the name of the corresponding C source file provided on the command line except that the extension **.a** replaces the extension **.c**. The output files contain assembly language code. This option may not be used in conjunction with the **o** option.

## The c Option

The **c** option tells the **cc** command to insert the comments generated by the C compiler during code generation into the assembly language file. The C compiler generates a comment at the beginning of each expression. It also generates comments for each variable declared in any given block. This type of comment contains the name of the variable and the value of its offset. The **c** option should only be used in conjunction with the **a** option.

## The D Option

The **D** option allows the user to define symbols on the command line as if they were defined in every one of the C source files with the preprocessor command *#define*. The syntax for this option is

> D=<*symbol*>[=*def*]

where <*symbol*> is the name of a symbol defined for the C preprocessor, which is replaced by *def* in the source code. If the user provides no definition, the value of <*symbol*> is 1. The definition is valid for all source files on the command line. The symbol is redefined at the beginning of each source file. A user who does not wish to include a definition in a particular source file can exclude it by using the preprocessor command *#undef* in that file. The **D** option may be used repeatedly on the command line.

## The f Option

When the **f** option is in effect, the **cc** command produces an output module suitable for firmware. In such a case the compiler does not allow any globally initialized data. It places all code and strings in the text segment and all global variables in the bss segment.

## The i Option

The **i** option specifies a directory to search for *#include* files. The syntax for this option is:

> **i**=<*dirname*>

where <*dirname*> is the name of a directory to search. The **i** option may be used repeatedly on the command line. The directories specified with the **i** option are searched in the order in which they appear on the command line.

The overall search for *#include* files proceeds as follows:

1. Search the directory containing the source file.
2. Search the current working directory.
3. Search the directories specified by the **i** option.
4. Search the directory *include* in the working directory.
5. Search the directory */lib/include.*

If the user encloses the file name used as an argument to the *#include* command in angle brackets, '<' and '>', the compiler does not search the directory containing the source file. If the file name specified begins with a slash character, '/', the compiler does not search any directories, but rather uses the file so specified as the *#include* file.

## The I Option

The **I** option instructs the **cc** command to stop when the first pass of the C compiler is complete. The name of each output file is the same as the name of the corresponding C source file provided on the command line except that the extension **.i** replaces the extension **.c**. The output files contain intermediate language, which cannot be read by the c compiler.

## The l Option

By default, the **cc** command passes to the linking-loader the names of the C libraries that contain standard I/O and math functions (*/lib/clibs* or */lib/clib* or both, depending on the hardware). The linking-loader searches these files when it tries to resolve external references. By invoking the **l** option, the user can specify the name of a library to search before searching these standard libraries. The syntax for this option is:

> **l**=<*lib_name*>

where <*lib_name*> is the name of a library to search. The **l** option may be used a maximum of 11 times on the command line. The libraries are searched in the order that the user specifies them.

## The L Option

The **L** option instructs the compiler to send to standard output a listing of each file specified on the command line that contains C source code. These listings, which contain line numbers, include listings of any *#include* files (see the *N* option for more details).

## The m Option

The **m** option tells the compiler to print the load and module maps from the linking-loader to standard output. These maps are explained in detail in the *4400 Assembly Language Reference* manual.

## The M Option

The **M** option instructs the compiler to compile, assemble, and link the source files specified on the command line and to produce as output one relocatable module. By default, the name of this file is **output.r**.

## The n Option

The **n** option instructs the **cc** command to stop when the first pass of the C compiler is complete. Pass 1 performs a syntactical check of the C source code in the files specified on the command line but generates no code.

## The N Option

The **N** option instructs the compiler to send to standard output a listing of each file specified on the command line which contains C source code. These listings, which contain line numbers, do not include listings of any *#include* files (see the *L* option for more details).

## The o Option

The **o** option specifies the name of the file containing the executable (or if the **M** option is in effect, the relocatable) output file. The syntax for this option is:

o=<*filename*>

The **o** option cannot be used in conjunction with the **r** or the **a** option.

## The O Option

The **O** option instructs the **cc** command to call the assembly language optimizer. Because it makes certain assumptions about the source files it reads and because it replaces these files with its optimized code, the optimizer should not be used on files containing hand-written assembly language source code. For these reasons, even if the user specifies the **O** option, the **cc** command does not run the optimizer on assembly language source files specified on the command line.

## The q Option

The C language requires that all items of type *char* and *short* be converted to *int* before any operations are performed on them. The **q** option bypasses this rule, allowing the generation of better code in many instances. In general, the code generated with the **q** option is smaller but equivalent to the code generated without the **q** option. For example, the statement

```
ch1 = ch2 << 3;
```

where **ch1** and **ch2** are of type *char*, generates code following these steps:

1. Convert *ch2* to type *int* (sign extend).
2. Shift the result of the conversion left 3 places.
3. Convert the result of the shift to type *char*.
4. Assign the result of step 3 to *ch1*.

In this example the conversions have no meaning. Because the C language ignores overflow, the code generated without the conversions has exactly the same effect.

The user should, however, be careful not to use the **q** option when overflow is expected to occur and is necessary to the operation being performed because the resulting code is not equivalent to that generated without the **q** option. For example, if the previous statement is changed to read

```
int1 = ch2 << 3;
```

where **int1** is of type *int*, use of the **q** option could cause the compiler to generate code that does not perform as expected, depending on what the user intended and what the value of *ch2* is. If the **q** option is in effect, the variable *ch2* is not converted to type *int* before the shift operation takes place. Any overflow from *ch2* is lost. If the **q** option is not in effect, *ch2* is converted to *int* before the shift operation takes place. Any overflow is retained for assignment to *int1*. An explicit cast of *ch2* into type *int* solves this problem.

In practice, using the **q** option does make the code smaller and faster, but it should be used cautiously. You should thoroughly debug a program before attempting to compile it with the **q** option. After compiling a program with the **q** option, the user should again check it thoroughly.

## The r Option

The **r** option instructs the compiler to produce a relocatable module for each input file, but not to call the linking-loader. The name of each output file is the same as the name of the corresponding file provided on the command line except that the extension **.r** replaces the extension **.c** or **.a**. The output files contain relocatable object code. This option may not be used with the o option.

## The R option

The **R** option tells the compiler to produce a relocatable module from each input file, to call the linking-loader to produce one executable output module, but not to delete the individual relocatable modules. The name of each relocatable module is the same as the name of the corresponding file provided on the command line except that the extension **.r** replaces the extension **.c** or **.a**.

## The t Option

The t option tells the compiler to produce as output a shared-text, executable module. This option is merely passed to the linking-loader. Shared-text files are discussed in detail in the *4400 Series Assembly Language Reference* manual.

## The U Option

The U option instructs the compiler to produce a line-feed character ($0A) for the C character constant *n* rather than the default of a carriage return ($0D).

## The v Option

The v option tells the **cc** command to send messages to standard error describing its activities. The messages show the command currently being executed, complete with the arguments and options sent to it.

## The w Option

The **w** option instructs the C preprocessor to warn the user about duplicate **#define** statements. Redefining a preprocessor variable is allowed, but it can make the debugging process very difficult.

## The x Option

The **x** option passes options directly to the linking-loader, **load**. The syntax for the **x** option is:

x=<*ldr_option*>

where <*ldr_option*> is some valid option to the **load** command. No plus sign, +, is allowed in front of <*ldr_option*>. For example, this use of the **x** option

**+x=F=/lib/nonstd_env**

specifies a file of options to the **load** command. As another example,

**+x=b=8M**

specifies the executable task to be 8 Mbytes in size.

## Examples

These examples illustrate some of the uses of the **cc** command:

```
cc test.c
```

This example compiles, assembles, and links the file **test.c**, producing as output the executable module **test**.

```
cc math.c float.c driver.c +o=testmath +Owsq
```

This example compiles the code in the three files specified on the command line, calls the assembly language optimizer, assembles the code, and calls the linking-loader. The output is the single executable module named **testmath**. The code generated by this command performs operations on variables of type *short* and *char* without converting them to integers. It does not check the availability of stack space before trying to obtain space on the stack. The compiler warns the user about duplicate definitions.

```
cc list.c +Ln
```

This example compiles the file **list.c** but generates no code. A listing of the C source file is sent to standard output.

```
cc games.c help.c +DDBG=1 +o=play +l=gamelib +t
```

This example compiles, assembles, and links the files **games.c** and **help.c**, producing as output the shared-text executable module *play*. The **D** option defines the variable *DBG*. The **l** option tells the linking-loader to search the library *gamelib* before it searches the standard C libraries.

```
cc prog.c +NSvqca
```

This example compiles the file **prog.c** and produces as output the assembly language file **prog.a**, which includes the comments generated by the compiler. A listing of the C source code is sent to standard output. This listing does not include the *#include* files. The code generated by this command performs operations on variables of type *short* and *char* without converting them to integers. It checks the availability of stack space before trying to obtain space on the stack. Because verbose mode is turned on, the command sends messages to standard error describing its activities.

## Description of the Language

Advanced features implemented by this version of the C language include the passing, returning, and assigning of structures and unions; *enumeration* types; and bit fields. The compiler supports the types *unsigned char, unsigned short,* and *unsigned long.*

## Object Sizes

Each variable defined in a C program requires some specific amount of space. Table 1-1 shows the sizes of the basic types of variables.

**Table 1-1**
*Variable Sizes*

| Type | Bytes |
|---|---|
| char | 1 |
| short | 2 |
| float | 4 |
| int | 4 |
| long | 4 |
| pointers | 4 |
| double | 8 |

The qualifier *unsigned*, which can be applied to variables of type *char, short, int,* or *long,* does not affect the size of the variable. *Short* implies *short int*; *long* implies *long int*; and *unsigned* implies *unsigned int.*

The types *float* and *double* conform to IEEE Task P754 proposed floating point standard for single and double precision formats respectively.

## Register Variables

A user on any system may apply the storage class *register* to variables of all basic types except *float* and *double.* Users whose hardware includes the MC68881 floating-point coprocessor may apply the storage class *register* to variables of all basic types. The compiler can honor the declarations of up to four pointer variables and five data variables (and, if applicable, five floating-point registers) as register variables per function. It changes the storage class of an invalid *register* declaration to *auto.*

## Section 2

# Kernighan and Ritchie Variations

## Introduction

This section describes the differences between Technical Systems Consultant´s C compiler, which is modeled after the UNIX System V C Compiler, and the language described by Kernighan and Ritchie in *Appendix A: C Reference Manual* of *The C Programming Language* (Kernighan and Ritchie, 1978). The numbers of the following sections and the accompanying page numbers correspond to the numbers appearing in that appendix.

## Identifiers (Names), page 179

External identifiers, which are used by the assembler and linking-loader, are restricted to:

```
59 characters,  2 cases
```

### Character Constants, page 180

An additional escape sequence is allowed for vertical tab.

```
vertical tab   VT     \v
```

### Enumeration Constants, new section

Names declared as enumerators are constants of the corresponding enumeration type and behave like integer constants.

## Hardware Characteristics, page 181

The hardware characteristics are:

| | |
|---|---|
| char | 8 bits |
| int | 32 bits |
| short | 16 bits |
| long | 32 bits |
| float | 32 bits |
| double | 64 bits |
| float range | 10E 38 |
| double range | 10E 308 |

## What's in a Name?, page 182

Each enumeration is conceptually a separate type with its own set of named constants. The properties of an enumeration type (*enum*) are identical to those of type *int*. The type *enum* is classified as an *integral* type.

The type *void* is used to specify an empty set of values. Its primary use is to define the type of a function that does not return a value.

The word *unsigned* may be used as an adjective to modify the types *char*, *short*, *int*, and *long*. Used by itself, *unsigned* is equivalent to *unsigned int*.

## Characters and Integers, page 183

Variables of type *char* range in value from -128 to 127 inclusive. Variables of the more explicit type, *unsigned char*, range in value from 0 to 255 inclusive.

## Void, new section

Objects declared to be type *void* may not be used in any way. Because a void expression denotes a nonexistent value, such an expression may only be used as an expression statement or as the left-hand operand of a comma expression. Expressions may be cast to type *void* in order, for example, to make explicit the discarding of the value of a function call used as an expression statement.

## Type Specifiers, page 193

The compiler supports two additional type specifiers:

```
"void"
enumeration specifier
```

The following three combinations are also supported:

*unsigned char*
*unsigned short*
*unsigned long*

## Structure, Union, and Enumeration Declarations, page 196

Fields are assigned from right to left. Fields are not signed, have only integral values, and should be declared *unsigned* although *int* is accepted.

Enumerations are unique types with named constants. The compiler treats enumeration variables and constants as being of type *int*. The syntax for the declaration of an enumeration type follows. Keywords are enclosed in quotation marks. Other words are descriptors that the user must replace with a specific example of the thing described.

```
enum specifier:
    "enum" { enum-list }
    "enum" identifier { enum-list }
    "enum" identifier
enum-list:
    enumerator
    enum-list, enumerator
enumerator:
    identifier
    identifier = constant-expression
```

The identifiers in an enumeration list are declared as constants and may appear wherever constants are allowed or required. The values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. These values can be altered by using an equals sign, ′=′, after an identifier—in which case the value of the constant is that specified after the equals sign. Subsequent identifiers continue the progression from the assigned value.

The names of all enumerators in the same scope must be distinct from each other. The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier. It names a particular enumeration. For example:

```
enum color { red, white, blue=10, orange };
enum color *colptr, colval;
colval = white;
colptr = &colval;
if ( *colptr == orange ) ...
```

This piece of code makes *color* the enumeration tag of a type describing various colors. The declarations declare *colval* as an object of that type and *colptr* as a pointer to an object of that type. The possible values are taken from the set {0,1,10,11}.

## Inclusion of an Information Field, new section

For operating systems such as UniFLEX, which support information fields in binary files, the preprocessor allows this command:

```
#info information-line
```

The information-line may be any text. All of the text, including the trailing carriage return, is placed in the information field of the binary file. This feature may not appear in all versions of the compiler because its usefulness is operating-system dependent.

## Structures and Unions, page 209

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Only identical structure and union types may be assigned.

## Explicit Pointer Conversions, page 210

The pointer representation for a 68000-based machine corresponds to a 32-bit integer and measures bytes. Variables of type *char* have no alignment requirements; variables of other types have even addresses. All aggregates, except arrays of characters, are also aligned on even addresses.

## Portability Considerations, page 211

The order of evaluation of the arguments to a function is not specified by the language. This compiler evaluates the arguments from right to left. Because character constants are really objects of type *int*, multicharacter constants are permitted. Up to four characters may be present in one constant.

## Anachronisms, page 212

A structure or union reference is a chain of member references (qualifications) prefixed either by a pointer to a structure or a union or by the name of a structure or a union. Because each qualification implies the addition of an offset within an address computation, older compilers (which failed to check for membership in the appropriate structure or union) allowed omission of those qualifications with an offset of 0. This compiler requires complete qualification.

# Section 3
# System Calls and Functions

This section contains manual pages for each C library system call and function. To make the manual pages easier to locate, they are listed in this section alphabetically and summarized below:

| | |
|---|---|
| **abort** | Send a task-abort signal to the current task, causing the task to stop immediately. |
| **abs** | Absolute value function. |
| **access** | Check the accessibility of a file. |
| **acct** | Begin or end system accounting. |
| **acos** | Arc-cosine function. |
| **addmount** | Add an entry to the system mount table. |
| **alarm** | Set the alarm clock of the task. |
| **asctime** | Generate a time stamp. |
| **asin** | Arc-sine function. |
| **atan** | Arc-tangent function. |
| **atan2** | Arc-tangent function. |
| **atof** | Convert a floating-point digit-string to a *double.* |
| **_atoh** | Convert a hexadecimal digit-string to a *long.* |
| **atoi** | Convert a decimal digit-string to an *int* |
| **atol** | Convert a string of decimal characters to an integer. |
| **_atoo** | Convert an octal digit-string to a *long.* |
| **_atos** | Convert a decimal digit-string to a *short.* |
| **basename** | Extract the simple filename from a pathname. |
| **brk** | Change the task´s data segment memory allocation. |
| **calloc** | Allocate memory. |
| **cdata** | Change the task´s data segment memory allocation. |
| **ceil** | Calculate the smallest integer not less than a certain specified value. |
| **chdir** | Change the working directory. |
| **chmod** | Change the access permissions of a file. |
| **chown** | Change the owner-ID of a file. |
| **chtim** | Change the modification date and time of a file. |
| **clearerr** | Clear the stream´s error-indicators. |
| **close** | Close an open file. |
| **closedir** | Close a directory-stream. |

| | |
|---|---|
| **control_pty** | Control a pseudo-terminal channel |
| **cos** | Calculate the cosine of an angle. |
| **cosh** | Calculate the hyperbolic cosine of a value. |
| **creat** | Create a new file or truncate an existing file. |
| **create_pty** | Create a pseudo-terminal channel |
| **_crypt** | Encrypt a character-string. |
| **ctime** | Generate a time stamp. |
| **daylight** | Daylight savings time flag. |
| **dirname** | Extract the directory prefix from a pathname. |
| **dup** | Duplicate a file descriptor. |
| **dup2** | Duplicate a file descriptor onto a specific file descriptor. |
| **ecvt** | Convert a floating-point value to a character-string. |
| **edata** | End-of-memory address of initialized data. |
| **end** | End-of-memory address of uninitialized data. |
| **endpwent** | End password-file handling. |
| **errno** | The system error code of the most recent system error. |
| **etext** | End-of-memory address of program text. |
| **execl** | Execute a program found in an executable binary file. |
| **execle** | Execute a program found in an executable binary file. |
| **execlp** | Execute a program found in an executable binary file. |
| **execv** | Execute a program found in an executable binary file. |
| **execve** | Execute a program found in an executable binary file. |
| **execvp** | Execute a program found in an executable binary file. |
| **exit** | Exit the program. |
| **_exit** | Exit the program. |
| **exp** | Calculate the exponential of a value. |
| **fabs** | Absolute value function. |
| **fclose** | Close a stream. |
| **fcntl** | Control the behavior of a file |
| **fcvt** | Convert a floating-point value to a character-string. |
| **fdopen** | Attach an open file to a stream. |
| **feof** | Test the end-of-file indicator of a stream. |
| **ferror** | Test the error-indicator of a stream. |

| | |
|---|---|
| **fflush** | Flush a stream opened for write access. |
| **fgetc** | Read a character from a stream. |
| **fgets** | Read a character-string from a stream. |
| **fileno** | Get a file descriptor for the file attached to a stream. |
| **finite** | Determine if a double precision floating point number is not an infinity. |
| **floor** | Calculate the largest integer not greater than a value. |
| **fmod** | Floating-point remainder function. |
| **fopen** | Open a file and attach it to a standard I/O stream. |
| **fork** | Create a new task. |
| **fprintf** | Write formatted data to a stream. |
| **fputc** | Write a character to a stream. |
| **fputs** | Write a character-string to a stream. |
| **fread** | Read data from a stream. |
| **free** | Free a block of allocated memory. |
| **freopen** | Reopen an open stream. |
| **frexp** | Separate the exponent from the mantissa of a floating-point value. |
| **fscanf** | Read and interpret formatted data from a stream. |
| **fseek** | Reposition a stream. |
| **fstat** | Get the status of an open file. |
| **ftell** | Get the current position of a stream. |
| **ftime** | Get the current time statistics for the operating system. |
| **_ftoa** | Convert a floating-point value to a character-string. |
| **ftw** | Descend the specified directory structure. |
| **fullname** | Generate the full pathname. |
| **fwrite** | Write data to a stream. |
| **gcvt** | Convert a floating-point value to a character-string. |
| **getc** | Read a character from a stream. |
| **getchar** | Read a character from the standard input stream. |
| **getcwd** | Get the pathname of the working directory. |
| **getenv** | Get information from the environment list. |
| **get_FPU_control** | Return the contents of the MC68881 control and status registers |
| **get_FPU_exception** | Access MC68881 coprocessor exception-information |

| | |
|---|---|
| **geteuid** | Get the effective user-ID number of the current task. |
| **getpass** | Get a password using a prompt. |
| **getpid** | Get task-ID number of the current task. |
| **getppid** | Get the task-ID number of the parent of the current task. |
| **getpw** | Get a password-file entry based on a user-ID. |
| **getpwent** | Get and decode the next entry in the system password file. |
| **getpwnam** | Get and decode the next entry in the system password file containing the given user-name. |
| **getpwuid** | Get and decode the next entry in the system password file containing the given user-ID number. |
| **gets** | Read a character-string from the standard input stream. |
| **getuid** | Get the user-ID number of the current task. login file that has specific <*ut_line*> value. |
| **getw** | Read a word from a standard I/O stream. |
| **gmtime** | Break down a system-time value into units in the Greenwich Mean Time zone. |
| **gtty** | Get the characteristics of an open character-device. |
| **idfd** | Return the last file descriptor which signalled "INPUT READY" |
| **_ierrmsg** | Initialize <*sys_errlist*> and <*sys_nerr*>. |
| **index** | Find the first occurrence of a character in a character-string. |
| **isalnum** | Determine if a value is an alphabetic character or a decimal digit. |
| **isalpha** | Determine if a value is an alphabetic character. |
| **isascii** | Determine if a value is an ASCII character. |
| **isatty** | Determine if a file descriptor references a character-special |
| **iscntrl** | Determine if a value is a control character. |
| **isdigit** | Determine if a value is a decimal digit. |
| **isgraph** | Determine if a value is a graphics character. |
| **islower** | Determine if a value is a lower-case alphabetic character. |
| **isnan** | Determine if a double precision floating point number is not-a-number. |
| **isprint** | Determine if a value is a printable character. |
| **ispunct** | Determine if a value is a punctuation character. |
| **isspace** | Determine if a value is a white-space character. |
| **isupper** | Determine if a value is an upper-case alphabetic character. |
| **isxdigit** | Determine if a value is a hexadecimal digit. |

| | |
|---|---|
| **_itostr** | Convert an *int* to a character-string. |
| **kill** | Send a signal to a task. |
| **_l2tos** | Convert two-byte integers to *short* integers. |
| **l3tol** | Convert three-byte integers to *long* integers. |
| **_l4tol** | Convert four-byte integers to *long* integers. |
| **ldexp** | Generate a floating-point value from a mantissa and an exponent. |
| **link** | Create a link to a file. |
| **localtime** | Break down a system-time value into units in the local time zone. |
| **lock** | Lock a task in memory or unlock a locked task. |
| **log** | Calculate the natural logarithm of a value. |
| **log10** | Calculate the base-10 logarithm of a value. |
| **longjmp** | Perform a non-local goto. |
| **lrec** | Add an entry to the lock table of the operating system. |
| **lseek** | Change the current file position of an open file. |
| **ltol3** | Convert *long* integers to three-byte integers. |
| **_ltol4** | Convert *long* integers to four-byte integers. |
| **_ltostr** | Convert a *long* to a character-string. |
| **make_realtime** | Declare the task to be a real-time task. |
| **malloc** | Allocate memory. |
| **matherr** | Floating-point error-handling function for built-ins. |
| **memccpy** | Copy memory. |
| **memchr** | Find a value in a block of memory. |
| **memcmp** | Compare two blocks of memory. |
| **memcpy** | Copy memory. |
| **memman** | Perform a memory management operation. |
| **memset** | Set a block of memory. |
| **mknod** | Add an entry to the file-system that is a directory, a character-special file, or a block-special file. |
| **mktemp** | Generate a unique pathname from a template. |
| **modf** | Separate a floating-point value into its integral and fractional parts. |
| **mount** | Mount a block-special file onto the file-system. |
| **nice** | Change the scheduling priority of a task. |
| **open** | Open an existing file. |

| | |
|---|---|
| **opendir** | Open a directory. |
| **pause** | Suspend the current task. |
| **pclose** | Close a stream connected to a pipe. |
| **perror** | Write a message explaining the error code in *errno*. |
| **pffinit** | Guarantee that the **cc** command loads the versions of standard I/O functions that contain floating-point conversions. |
| **phys** | Access or release a system resource. |
| **pipe** | Create a pipe. |
| **popen** | Open a pipe and attach it to a standard I/O stream. |
| **pow** | Raise a value to a power. |
| **printf** | Write formatted data to *stdout*. |
| **profil** | Start or stop monitoring the current task. |
| **putc** | Write a character to a stream. |
| **putchar** | Write a character to *stdout*. |
| **putenv** | Modify or add an environment-variable definition to the environment list. |
| **put_FPU_control** | Change the contents of the MC68881 control and status registers |
| **put_FPU_exception** | Update MC68881 coprocessor exception-information |
| **putpwent** | Format and write a system password-file record. |
| **puts** | Write a character-string to *stdout*. |
| **putw** | Write a word to a stream. |
| **qsort** | Sort data. |
| **rand** | Generate a random number. |
| **read** | Read data from an open file. |
| **readdir** | Read the next entry in an open directory. |
| **realloc** | Reallocate an allocated block of data. |
| **rewind** | Rewind a stream. |
| **rewinddir** | Rewind a directory-stream. |
| **rindex** | Find the last occurrence of a character in a character-string. |
| **rmvmount** | Remove an entry from the system mount table. |
| **rrand** | Set the seed of the random number generator to a value generated from the current system-time value. |
| **rump_create** | Create a new managed resource. |
| **rump_dequeue** | Relinquish access to a named resource. |

| | |
|---|---|
| **rump_destroy** | Destroy a managed resource. |
| **rump_enqueue** | Obtain exclusive access to a named resource. |
| **sbrk** | Change the memory allocation of the data segment. |
| **scanf** | Read and interpret formatted data from *stdin*. |
| **seekdir** | Change the current position of a directory-stream. |
| **set_ftm** | Change the last-modification time of a file. |
| **set_high_address_mask** | Set the hardware high address mask register |
| **setbuf** | Set buffering attributes of a stream. |
| **setjmp** | Setup for a non-local goto. |
| **setpwent** | Reset password-file handling. |
| **setuid** | Change both the user-ID and the effective user-ID. |
| **signal** | Change the signal-handling address for a specific signal in the current task. |
| **sin** | Calculate the sine of an angle. |
| **sinh** | Calculate the hyperbolic sine of a value. |
| **sleep** | Suspend execution for an interval. |
| **sprintf** | Generate a character-string containing formatted data. |
| **sqrt** | Calculate the square root of a value. |
| **srand** | Set the seed of the random number generator. |
| **sscanf** | Interpret formatted data from a character-string. |
| **stack** | Check and expand memory allocated to the stack segment of the task. |
| **stat** | Get the status of a file. |
| **stderr** | Standard error stream for standard I/O. |
| **stdin** | Standard input stream for standard I/O. |
| **stdout** | Standard output stream for standard I/O. |
| **stime** | Set the system-time value. |
| **_stol2** | Convert *short* integers to two-byte integers. |
| **strcat** | Concatenate one character-string onto another. |
| **strchr** | Find the first occurrence of a character in a character-string. |
| **strcmp** | Compare two character-strings. |
| **strcmpci** | Compare two character-strings (case insensitive). |
| **strcpy** | Copy a character-string. |
| **strcspn** | Determine the unlike character-count. |

| | |
|---|---|
| **strerror** | Return a pointer to a message describing the specified error number. |
| **strlen** | Determine the length of a character-string. |
| **strncat** | Concatenate one character-string onto another. |
| **strncmp** | Compare two character-strings. |
| **strncmpci** | Compare two character-strings (case insensitive). |
| **strncpy** | Copy a character-string. |
| **strpbrk** | Find the first occurrence of any of a list of characters in a character-string. |
| **strrchr** | Find the last occurrence of a character in a character-string. |
| **strspn** | Determine the like character-count. |
| **strstr** | Find a substring with a character-string. |
| **strstrci** | Find a substring within a character-string (case insensitive). |
| **_strtoi** | Convert the digits in a character-string to an *int*. |
| **strtok** | Extract the next token from a character-string. |
| **strtol** | Convert the digits in a character-string to a *long*. |
| **stty** | Set the characteristics of an open character-device. |
| **sync** | Update the file-system. |
| **sys_errlist** | This is a global table containing references to messages describing system error codes. |
| **sys_nerr** | The number of system error messages referenced by the global table *sys_errlist*. |
| **system** | Issue a shell command. |
| **time** | Get the current system-time value. |
| **times** | Get the CPU-usage information for the current task. |
| **timezone** | Current time zone value. |
| **toascii** | Generate a value that is within the range of valid ASCII characters. |
| **_tolower** | Convert an upper-case character to a lower-case character. |
| **_toupper** | Convert a lower-case character to an upper-case character. |
| **truncf** | Set the size of an open file. |
| **ttyname** | Generate the pathname for a terminal. |
| **ttyslot** | Get the terminal number of the controlling terminal for the task. |
| **tzname** | Time-zone name abbreviations. |
| **tzset** | Initialize external variables containing time parameters. |
| **umask** | Change the file-creation permissions mask for the task. |

| | |
|---|---|
| **umount** | Unmount a mounted device. |
| **ungetc** | Push a character onto an input stream. |
| **unlink** | Remove a link to a file. |
| **urec** | Remove an entry from the operating system lock table. |
| **utime** | Change the last-modification time for a file. |
| **vfork** | Create a new task. |
| **wait** | Suspend the task until a child task terminates. |
| **write** | Write data to an open file. |

## abort

Send a task-abort signal to the current task, causing the task to stop immediately.

## SYNOPSIS

```
void abort();
```

### Arguments

None

### Returns

Never

## DESCRIPTION

**Abort** sends a task-abort signal, #<n>, to the current task, which causes the task to terminate immediately. The task-abort signal cannot be caught or ignored. The function never returns to the caller.

The system signals can be found in the *kill()* manual page.

## ERRORS REPORTED

None

## NOTES

The termination status received by the parent of the current task contains an exit code of zero, a termination code indicating that the task terminated because of a task-abort signal, and a flag that indicates if a core-image file was produced.

## SEE ALSO

System Call: *signal(), wait()*

Command: *int*

## abs

Absolute value function.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| *<i>* | The number whose absolute value is to be calculated |

### Returns

The absolute value of the argument *<i>*

## DESCRIPTION

**Abs** calculates the absolute value of the argument *<i>*. It returns the calculated value as its result.

## NOTES

If *<i>* is the largest negative number, **abs()** returns that value as its result.

# access

Check the accessibility of a file.

## SYNOPSIS

```
#include <errno.h>
int access(path, perms)
     char     *path;
     int       perms;
```

### Arguments

| | |
|---|---|
| <path> | The *<path>* argument is a character-string that specifies the directory location of the file. *Access* locates the file to be checked by following the specified path. |
| <perms> | A value indicating the type of access to check |

### Returns

This function returns a zero if access is permitted, otherwise the function returns a **-1** with *<errno>* set to the system error code (this indicates the reason for denying access).

## DESCRIPTION

The **access** function checks the permissions of the file reached by the pathname in the character-string referenced by *<path>*. The value *<perms>* specifies the type of permission to check. If the file exists, the function returns zero and grants the requested access. Otherwise, this function returns **-1** with *<errno>* to indicate the reason the access is denied.

A **-1** returned value indicates the path could not be followed, a part of the path is not a directory, the pathname does not reach a file, or the file does not grant the effective user the requested access permissions.

The value *<perms>* is a bit-string that tells the *access* function the types of permissions to check. *<Perms>* may be any combination of these values:

| | |
|---|---|
| 0x01 | Read |
| 0x02 | Write |
| 0x04 | Execute (search) |

A *<perms>* value of zero tells the function to check the path to the file to see if the file exists.

## Errors Reported

| | |
|---|---|
| EACCES | The file permissions do not grant the requested access type |
| EMSDR | Cannot follow the path to the file |
| ENOEP | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

If the current effective user is the owner of the specified file, the *access* function checks the file permissions for its owner. Otherwise, it examines the permissions granted for users other than its owner.

## SEE ALSO

System Call: *chmod(), stat()*

# acct

Begin or end system accounting.

## SYNOPSIS

```
#include <errno.h>
#include <sys/acct.h>
int acct(path)
     char     *path;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string that contains a pathname for the file where to write accounting records, or (*char* *) NULL |

### Returns

Zero if successful, otherwise *-1* with <*errno*> set to the system error code

## DESCRIPTION

If <*path*> is not *(char *) NULL*, the **acct** function begins system accounting. While system accounting is active, every time a task terminates the system writes a system accounting record (described later) to the file reached by the pathname referenced by <*path*> . The referenced file must already exist. If <*path*> is *(char *) NULL*, the **acct** function ends active system accounting, if any.

This function returns zero if it successfully performs its function, otherwise it returns *-1* with <*errno*> set to the system error code. This function requires that the current effective user-ID be that of the system manager.

The function fails if <*path*> is not *(char *) NULL* and the path in the pathname can not be followed, a part of the path is not a directory, the pathname does not reach a file, or system accounting is already active. The function also fails if the current effective user is not the system manager.

The following structure describes the record written by the system to the specified file each time a task terminates.

```
struct acct
{
     short          ac_uid;
     long           ac_strt;
     long           ac_end;
     char           ac_syst[3];
     char           ac_usrt[3];
     unsigned int   ac_stat;
     char           ac_tty;
     char           ac_mem;
     unsigned int   ac_blks;
     char           ac_spare[2];
     char           ac_name[8];
};
```

The *ac_uid* entry contains the user-ID number associated with the task.

*ac_strt* contains the system-time at the start of the task.

*ac_end* contains the system-time at the end of the task.

*ac_syst* (a three-byte integer) contains the number of CPU-seconds used by the system on behalf of the task.

*ac_usrt* (a three-byte integer) contains the number of CPU-seconds used by the task.

*ac_stat* contains task's termination status.

*ac_tty* contains the task's controlling terminal number.

*ac_mem* contains the maximum number of 1028-byte blocks of memory ever allocated to the task at one time.

*ac_blks* contains the number of I/O units used by the task.

*ac_spare* is currently unused.

*ac_name* contains the first eight characters of the command that initiated the task.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EEXIST | System accounting is already active |
| EMSDR | Could not follow the path to the file |
| ENOEP | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The **acct** function does not report an error if the *<path>* is *(char *) NULL* and system accounting is not currently active.

The operating system writes accounting records to the end of the specified file.

## SEE ALSO

Command: */etc/sysact*

## acos

Arc-cosine function.

## SYNOPSIS

```
#include <math.h>
double acos(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The cosine value to use to compute an angle |

### Returns

The angle, in radians, that has the cosine *<x>*

## DESCRIPTION

The **acos** function calculates the angle in radians between 0.0 and pi that has as its cosine the value *<x>*.

The function expects *<x>* to be between -1.0 and 1.0 inclusive. Values outside of that range cause a domain error. If the function detects a domain error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. It sets the *<type>* element of the structure to **DOMAIN**, *<name>* to the address of the character-string **acos**, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the function writes the message

```
acos() error:  Argument is out of range
```

to the standard I/O stream *<stderr>* and sets *<errno>* to **EDOM**. If **matherr()** returns something other than 0, it returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *asin(), atan(), cos(), matherr()*

# addmount

Add an entry to the system mount table.

## SYNOPSIS

```
void *addmount(device, path)
     char     *device;
     char     *path;
```

### Arguments

| | |
|---|---|
| <device> | The address of a character-string containing the pathname of the device which is mounted |
| <path> | The address of a character-string containing the pathname of the directory on which the device is mounted |

### Returns

Void

## DESCRIPTION

This function adds an entry to the system´s mount-table file. The entry is composed of the pathname of the device, the pathname of the directory, the actual user-ID of the current task, and the current time.

If there already is an entry in the system´s mount-table file with the same device pathname, that entry is overwritten; otherwise, a new entry is created.

## NOTES

The **addmount()** function does not perform an actual mount of the device on the directory; it only manipulates the system´s mount-table file.

No error is reported if the system´s mount-table file does not exist.

If the device pathname does not begin with a ´/´, the string /dev/ is prepended to the specified pathname before the system´s mount-table file is searched.

## SEE ALSO

C Library: *rmvmount()*

System Call: *mount(), umount()*

Command: */etc/mount, /etc/unmount*

## alarm

Set the alarm clock of the task.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <sec> | The number of seconds to elapse before sending an alarm signal to the current task |

### Returns

The number of seconds remaining from a previous alarm clock request (zero if none)

## DESCRIPTION

If *<sec>* is not zero, the **alarm** function arms the alarm clock of the task so the system sends an alarm signal to the current task after the specified number of seconds has elapsed. If the alarm clock was already armed, the **alarm** function cancels the previous alarm clock request. If *<sec>* is zero, the **alarm** function cancels the previous alarm clock request.

This function returns as its result the number of seconds remaining on a previous alarm clock request, or zero if there was no previous request.

## ERRORS REPORTED

None

## NOTES

An alarm signal causes the current task to terminate unless it explicitly catches or ignores alarm signals.

The actual amount of time that elapses before the system sends the alarm signal may be slightly less than the requested time, since the system tics occur on one-second intervals.

## SEE ALSO

C Library: *sleep()*

System Call: *pause(), signal(), wait()*

Command: *sleep*

## asctime

Generate an ASCII time stamp.

## SYNOPSIS

```
#include <time.h>
char *asctime(dttm)
     struct tm     *dttm;
```

### Arguments

| | |
|---|---|
| <dttm> | The address of a structure containing date and time information |

### Returns

The address of the generated ASCII time stamp

## DESCRIPTION

The **asctime** function generates an ASCII time stamp that represents the date and time information in the structure referenced by *<dttm>*. It returns the address of the time stamp as its result.

A time stamp is a 26-character string of characters (including the terminating null-character) that represents:

- the day of the week
- the month of the year
- the day of the month
- the hour
- minute
- second
- year

The time stamp is generated by the *sprintf()* format:

```
"%3s %3s %2.2d %2.2d:%2.2d:%2.2d %4.4d\n"
```

## NOTES

The character-string referenced by the result of this function is in static memory and is overwritten by subsequent calls to this function and **ctime()**.

## SEE ALSO

C Library: *ctime(), gmtime(), localtime(), sprintf()*

System Call: *time()*

Command: *date*

# asin

Arc-sine function.

## SYNOPSIS

```
#include <math.h>
double asin(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The sine value used to compute an angle |

### Returns

The angle, in radians, that has the sine *<x>*

## DESCRIPTION

The **asin** function calculates the angle in radians between -pi/2 and pi/2 that has a sine value of *<x>*. It returns that angle as its result.

The **asin** function expects the value *<x>* to be between -1.0 and 1.0 inclusive. Values outside of that range cause a domain error. If the function detects a domain error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. It sets the *<type>* element of the structure to **DOMAIN**, *<name>* to the address of the character-string *asin*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the function writes the message

```
asin() error:  Argument is out of range
```

to the standard I/O stream *<stderr>* and sets *<errno>* to **EDOM**. If **matherr()** returns something other than zero, it returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *acos(), atan(), matherr(), sin()*

## atan

Arc-tangent function.

## SYNOPSIS

```
#include <math.h>
double atan(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The tangent value used to compute an angle |

### Returns

The angle, in radians, that has the tangent *<x>*

## DESCRIPTION

The **atan** function calculates the angle in radians between -pi/2 and pi/2 that has as its tangent the value *<x>*. **Atan** returns that angle as its result.

## SEE ALSO

C Library: *acos(), asin(), atan2(), tan()*

# atan2

Arc-tangent function.

## SYNOPSIS

```
#include <math.h>
double atan2(x, y)
     double    x;
     double    y;
```

### Arguments

| | |
|---|---|
| <x> | The dividend of the tangent value used to compute an angle |
| <y> | The divisor of the tangent value used to compute an angle |

### Returns

The angle, in radians, that has the tangent *<x>*/*<y>*

## DESCRIPTION

The **atan2** function calculates the angle (in radians) between -pi and pi that has as its tangent the value *<x>*/*<y>*. **Atan2** returns that angle as its result. This function has twice the range of the **atan()** function because it takes into account the signs of values defining the tangent of the angle. It also handles a divisor *<y>* of zero so that no zero division error occurs.

This function permits *<x>* and *<y>* to be any value, as long as they are not both 0.0. Having both arguments 0.0 causes a singularity error. If the function detects a singularity error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. **Atan2** sets the *<type>* element of the structure to **SING**, *<name>* to the address of the character-string *atan2*, *<arg1>* to *<x>*, and *<arg2>* to *<y>*.

If **matherr()** returns **0**, the function writes the message

```
atan2() error:  Both arguments are 0.0
```

to the standard I/O stream *<stderr>* and sets *<errno>* to **EDOM**. If **matherr()** returns something other than zero, it returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C library: *acos(), asin(), atan(), matherr(), tan()*

# atof

Convert a floating-point digit-string to a *double.*

## SYNOPSIS

```
double atof(str)
     char     *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The floating-point value generated

## DESCRIPTION

The **atof** function generates a *double* from the character-string referenced by *<str>*. It returns the generated value as its result.

The **atof** function expects the character-string to contain optional whitespace (see **isspace**()), which it ignores, followed by an optional signed string of decimal digits (see **isdigit**()) containing an optional decimal point, followed by an optional exponent. The exponent consists of an 'E' or 'e' character followed by an optional sign followed a string of optional decimal digits. It continues converting until it reaches the end of the string or it finds an inappropriate character.

## NOTES

The function returns the properly signed maximum value if the character-string represents a value whose magnitude is larger than can be represented by a *double.*

## SEE ALSO

C Library: *_atoh(), atoi(), _atoo(), atol(), _atos(), ecvt(),fcvt(), _ftoa(), gcvt()*

# _atoh

Convert a hexadecimal digit-string to a **long**.

## SYNOPSIS

```
long _atoh(str)
     char     *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The integer generated from the character-string referenced by *<str>*

## DESCRIPTION

The **_atoh** function generates a *long* from the character-string referenced by *<str>*. It returns that value as its result.

The function expects the character-string to contain optional whitespace (see **isspace()**), which is ignored, followed by an optional sign, followed by a optional ('0') and an ('x') or ('X'), which are ignored, followed by a string of hexadecimal digits (see **isxdigit()**). **_Atoh** continues converting until it reaches the end of the string or it finds inappropriate character.

## NOTES

The function ignores overflow errors.

The conversion is performed by:

```
strtol(str, (char **) NULL, 16)
```

## SEE ALSO

C Library: *atof(), atoi(), _atoo(), atol(), _atos(), strtol()*

## atoi

Convert a decimal digit-string to an *int*.

## SYNOPSIS

```
int atoi(str)
     char     *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The integer generated from the character-string referenced by *<str>*

## DESCRIPTION

The **atoi** function generates an *int* from the character-string referenced by *<str>*. **Atoi** returns the generated value as its result.

The **atoi** function expects the character-string to contain optional whitespace (see **isspace()**), which is ignored, followed by an optionally signed string of decimal digits (see **isdigit()**). **Atoi** continues converting until it reaches the end of the string or it finds an inappropriate character.

## NOTES

Overflow errors are ignored.

The conversion is performed by:

```
(int) strtol(str, (char **) NULL, 10)
```

## SEE ALSO

C Library: *atoff(), _atoh(), _atoo(), atol(), _atos(), strtol()*

# atol

Convert a string of decimal characters to an integer.

## SYNOPSIS

```
    l(str)
     char      *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The integer generated from the character-string referenced by <str>

## DESCRIPTION

This **atol** function generates a *long* from the character-string referenced by *<str>*. It returns that value as its result. The **atol** function expects the character-string to contain optional whitespace (see **isspace()**), which is ignored, followed by a string of decimal digits (see **isdigit()**). The function converts until it reaches the end of the string or it detects an inappropriate character.

## NOTES

Overflow errors are ignored. The conversion is performed by

```
        strtol(str, (char **) NULL, 10)
```

## SEE ALSO

C Library: *_atoh(), atoi(), _atoo(), _atos(), strtol()*

# _atoo

Convert an octal digit-string to a *long*.

## SYNOPSIS

```
long _atoo(str)
     char     *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The integer generated from the character-string referenced by *<str>*

## DESCRIPTION

The **_atoo** function generates a *long* from the character-string referenced by *<str>*. **_Atoo** returns that value as its result.

The **_atoo** function expects the character-string to contain optional whitespace (see **isspace()**), which is ignored, followed by an optional sign, followed by a string of octal digits (digits 0 through 7). The function continues until it reaches the end of the string or it finds an inappropriate character.

## NOTES

Overflow errors are ignored.

The conversion is performed by

```
strtol(str, (char **) NULL, 8)
```

## SEE ALSO

C Library: *atof(), _atoh(), atoi(), atol(), _atos(), strtol()*

## _atos

Convert a decimal digit-string to a *short.*

## SYNOPSIS

```
short _atos(str)
     char     *str;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert |

### Returns

The integer generated from the character-string referenced by *<str>*

## DESCRIPTION

The *_atos* function generates a *short* from the character-string referenced by *<str>*. It returns that value as its result.

The *_atos* function expects the character-string to contain optional whitespace (see **isspace()**), which is ignored, followed by an optionally signed string of decimal digits (see **isdigit()**). The function converts until it reaches the end of the string or it finds an inappropriate character.

## NOTES

Overflow errors are ignored.

The conversion is performed by

```
        (short) strtol(str, (char **) NULL, 10)
```

## SEE ALSO

C Library: *atof(), _atoh(), atoi(), atol(), _atoo(), strtol()*

# basename

Extract the simple filename from a pathname.

## SYNOPSIS

```
char *basename(path, suffix)
     char     *path;
     char     *suffix;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname |
| <suffix> | The address of a character-string containing a filename *<suffix>* or *(char *) NULL* if none |

### Returns

The address of a character-string containing the simple filename

## DESCRIPTION

The **basename** function removes the directory prefix, if any, from the pathname in the character-string referenced by *<path>*. If *<suffix>* is not *(char *) NULL*, the **basename** function also removes the characters in the character-string referenced by *<suffix>* from the end of the pathname, if the pathname ends in those characters. The **basename** function returns as its result the address of a character-string containing the extracted simple filename.

## NOTES

The result of the **basename** function is in static memory and is overwritten by subsequent calls to this function.

**Basename** does not check the validity of the *<path>*. Nor does it verify that the *<path>* exists on the filesystem.

## SEE ALSO

C Library: *dirname(), fullname()*

Command: *basename*

## brk

Change the task's data segment memory allocation.

## SYNOPSIS

```
#include <errno.h>
int brk(addr)
     char      *addr;
```

### Arguments

| | |
|---|---|
| <addr> | The requested end-of-segment address for the data segment |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **brk** function changes the amount of memory allocated to the data segment so that the end-of-segment address of the data segment is *<addr>*. If the function succeeds, it returns zero as its result. Otherwise, it returns **-1** with *<errno>* set to the system error code describing the reason for the failure of the function.

The function fails if the address specified in *addr* is less than the lowest address in the data segment, or if it could not allocate enough memory to satisfy the request.

If the requested end-of-segment address is higher than the current end-of-segment address of the data segment, the **brk** function allocates memory to the segment. If the requested end-of-segment address is lower than the current end-of-segment address of the data segment, the **brk** function releases memory from the segment.

## ERRORS REPORTED

| | |
|---|---|
| ENOMEM | Not enough memory is available |

## NOTES

The end-of-segment address of a segment is the lowest logical address that is higher than the highest logical address of memory allocated to the segment.

## SEE ALSO

C Library: *calloc(), EDATA, free(), malloc(), realloc()*

System Call: *cdata(), sbrk()*

## calloc

Allocate memory.

## SYNOPSIS

```
char *calloc(num, size)
     unsigned  num;
     unsigned  size;
```

### Arguments

| | |
|---|---|
| <num> | The number of units to allocate |
| <size> | The size of a unit |

### Returns

The address of the allocated block of memory or *(char *) NULL* if no memory is available.

## DESCRIPTION

The **calloc** function allocates *<num>* times *<size>* bytes of memory from the area of available memory. **Calloc** returns the address of the first byte of the allocated memory or *(char *) NULL* if no memory is available.

The first byte of the allocated memory is aligned for any use.

## NOTES

Return allocated memory to the arena of available memory by using **free**().

## SEE ALSO

C Library: *free(), malloc(), realloc()*

System Call: *brk(), cdata(), sbrk()*

# cdata

Change the task´s data segment memory allocation.

## SYNOPSIS

```
#include <errno.h>
int cdata(addr)
     char     *addr;
```

### Arguments

| | |
|---|---|
| <addr> | The requested end-of-segment address for the data segment |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **cdata** function changes the amount of memory allocated to the data segment so the end-of-segment address is *<addr>*. If **cdata** allocates memory to the data segment, it allocates memory that is physically contiguous to the last page of memory allocated to that segment. If **cdata** succeeds, it returns zero as its result. Otherwise, **cdata** returns **-1** with *<errno>* set to the system error code describing the reason for failure.

**Cdata** fails if the address *<addr>* is less than the lowest address in the data segment, or if it could not allocate enough contiguous memory to satisfy the request.

If the requested end-of-segment address is higher than the current end-of-segment address, **cdata** allocates memory to the segment that is physically contiguous to the last page of the segment. If the requested end-of-segment address is lower than the current end-of-segment address, the function releases memory from the segment.

## ERRORS REPORTED

| | |
|---|---|
| ENOMEM | Not enough memory is available |

## NOTES

The end-of-segment address is the lowest logical address that is higher than the highest logical address of memory allocated to the segment.

On virtual memory systems, **cdata** is functionally equivalent to the **brk()** function.

## SEE ALSO

C Library: *calloc(), EDATA, free(), malloc(), realloc()*

System Call: *brk(), sbrk()*

# ceil

Calculate the smallest integer not less than a specified value.

## SYNOPSIS

```
#include <math.h>
double ceil(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The floating-point argument to the function |

### Returns

The smallest integer that is not less than *<x>*

## DESCRIPTION

The **ceil** function calculates the smallest integer that is not less than the value *<x>*. As a result, **ceil** returns that value represented as a *double.*

## SEE ALSO

C Library: *floor()*

# chdir

Change the working directory.

## SYNOPSIS

```
#include <errno.h>
int chdir(path)
     char     *path;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname to the new working directory |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **chdir** function changes the working directory to the directory reached by the pathname in the character-string referenced by *<path>*. **Chdir** returns zero as its result if it successfully changes the working directory to the specified directory. Otherwise, **chdir** returns **-1** with *<errno>* set to the system error code.

The **chdir** function fails if the pathname could not be followed or a part of the pathname is not a directory.

## ERRORS REPORTED

| | |
|---|---|
| EMSDR | could not follow the path to this file |
| ENOTDIR | A part of the path is not a directory or the file reached by the pathname is not a directory |

## SEE ALSO

C Library: *getcwd()*

Command: *chd*

# chmod

Change the access permissions of a file.

## SYNOPSIS

```
#include <errno.h>
#include <sys/modes.h>
int chmod(path, perms)
     char     *path;
     int       perms;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname to the file whose access permissions you want to change |
| <perms> | A bit-string describing the permissions to set on the file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **chmod** function changes the access permissions of the file reached by the pathname in the character-string referenced by *<path>* to those described by the bit-string *<perms>*. The **chmod** function requires that the current effective user be the owner of the file or the system manager. **Chmod** returns zero as its result if it successfully changes the access permissions of the file. Otherwise, **chmod** returns **-1** with *<errno>* set to the system error code.

**Chmod** fails it could not follow the path, a file in the path is not a directory, the pathname does not reach a file, or the current effective user is not the owner of the file or the system manager.

The value *<perms>* is a bit-string describing the permissions to set on the file. The include-file *sys/modes.h* defines these constants that describe the meanings of each bit used by the function in the bit-string:

```
S_IREAD    0x01
S_IWRITE   0x02
S_IEXEC    0x04
S_IOREAD   0x08
S_IOWRITE  0x10
S_IOEXEC   0x20
S_ISUID    0x40
```

**S_IREAD** grants reading permission to the owner, **S_IWRITE** grants writing permission to the owner, and **S_IEXEC** grants searching permission to the owner (if this is a directory; otherwise **S_IEXEC** grants execution permission). **S_IOREAD** grants reading permission to users other than the owner of the file, **S_IOWRITE** grants writing permission to others, and **S_IOEXEC** grants searching permission to others if this is a directory, or execution permission if this is a file. **S_ISUID** causes the effective user-ID to change to the owner of the file when the program in the file is executed. The results of **chmod** are undefined if bits other than those defined above are set in the bit-string *<perms>*.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager or file owner |
| EMSDR | could not follow the path to this file |
| ENOENT | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## SEE ALSO

System Call: *chown(), fstat(), stat()*

Command: *dir, perms*

## chown

Change the owner-ID of a file.

## SYNOPSIS

```
#include <errno.h>
int chown(path, uid)
     char     *path;
     int       uid;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname to the file whose owner-ID you want to change |
| <uid> | The user-ID to be the new owner-ID of the file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **chown** function changes the owner-ID of the file reached by the pathname in the character-string referenced by *<path>* to the value *<uid>*. The owner-ID of a file is the user-ID of the user that is the owner of the file. The **chown** function requires that the current effective user be the system manager. **Chown** returns zero as its result if it successfully changes the owner-ID of the specified file. Otherwise, it returns **-1** with *<errno>* set to the system error code.

**Chown** fails if it could not follow the path, the path contains a file that is not a directory, the path does not reach a file, or the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | the current effective user is not the system manager |
| EMSDR | could not follow the path to this file |
| ENOENT | the pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The user-ID *uid* does not have to be in the system password file.

## SEE ALSO

System Call: *chmod(), fstat(), stat()*

Command: *dir, owner*

## chtim

Change the modification date and time of a file.

## SYNOPSIS

```
#include <errno.h>
int chtim(path, time)
     char     *path;
     long      time;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string that contains a pathname to the file whose date and time you want to to change |
| <time> | The system-time value to set as the new date and time for the file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **chtim** function changes the modification date and time of the file reached by the pathname in the character-string referenced by *<path>* to the system-time value *<time>*. The **chtim** function can not change the modification date and time of a file that is currently open by another task. **Chtim** also expects the current effective user to be the system manager. The function returns zero as its result if it successfully changes modification date and time of the specified file. Otherwise, **chtim** returns **-1** with *<errno>* set to the system error code.

The **chtim** function fails if it could not follow the path, the path contains a file that is not a directory, the path does not reach a file, the file is currently open by another task, or the current effective user is not the system-manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The effective current user is not the system manager |
| EBSY | The specified file is currently open by another task |
| EMSDR | Could not follow the path to this file |
| ENOENT | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The **chtim** function does not demand that <*time*> be between the creation date of the file and the current time-of-day.

The system represents time as the number of seconds that has elapsed since the epoch. The system defines the epoch as 00:00 (midnight), January 1, 1980, Greenwich Mean Time.

Other functions which change a file´s modification date and time are **chmod()**, **chown()**, **creat()**, **link()**, **open()**, and **unlink()**.

## SEE ALSO

System Call: *chmod(), chown(), creat(), link(), open(), unlink()*

Command: **touch**

# clearerr

Clear the stream´s error-indicators.

## SYNOPSIS

```
#include <stdio.h>
int clearerr(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream |

### Returns

Undefined

## DESCRIPTION

The **clearerr** function clears (resets) the error-indicator and the end-of-file indicator on the standard I/O stream referenced by *<stream>*.

## NOTES

The **clearerr** function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once.

The **ferror()** function tests the error-indicator of a stream.

The **feof()** function tests the end-of-file indicator of a stream.

## SEE ALSO

C Library: *fdopen(), feof(), ferror(), fopen(), stderr, stdin, stdout*

# close

Close an open file.

## SYNOPSIS

```
#include <errno.h>
int close(fildes)
     int  fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the file to close |

### Returns

Zero if successful, otherwise *-1* with *<errno>* set to the system error code

## DESCRIPTION

The **close** function closes the file referenced by the file descriptor *<fildes>*. The **close** function returns zero as its result if it successfully closes the file, otherwise it returns **-1** with *<errno>* set to the system error code.

**Close** fails if *<fildes>* is out of range or does not reference an open file.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode. |
| EINVAL | An argument to the function is invalid. |

## NOTES

When a task terminates, the system automatically closes all files that the task has open.

## SEE ALSO

C Library: *fclose(), fopen()*

System Call: *creat(), dup(), dup2(), open(), pipe()*

## closedir

Close a directory-stream.

## SYNOPSIS

```
#include <sys/dir.h>
void closedir(pdir)
     DIR      *pdir;
```

### Arguments

| | |
|---|---|
| <pdir> | A reference to a directory-stream |

### Returns

Void

## DESCRIPTION

The **closedir** function closes the directory-stream referenced by *pdir*. **Closedir** closes the directory attached to the directory-stream and releases all of the resources allocated to that directory stream.

## NOTES

The include-file *sys/dir.h* contains definitions for the data types, structures, constants, and functions needed to read directories.

## SEE ALSO

C Library: *opendir(), readdir(), rewinddir(), seekdir(), telldir()*

# control_pty

Control a pseudo-terminal channel

## SYNOPSIS

```
#include <errno.h>
#include <sys/pty.h>
int control_pty(fd, fcn, cval)
    int fd;
    int fcn;
    int cval;
```

### Arguments

| | |
|---|---|
| <fd> | A file descriptor for master mode access of a pseudo-terminal |
| <fcn> | A function code |
| <cval> | A control value |

### Returns

The current state of the pseudo-terminal if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **control_pty** function is used to control the behavior of a pseudo-terminal channel. All functions return the state of the channel as described for the **PTY_INQUIRY** function.

The **PTY_INQUIRY** function is used to return the state of the channel. For this function, *cval* is ignored. The value returned is a combination of bits which describe the state of the channel. The bits are:

| | |
|---|---|
| PTY_PACKET_MODE | Reads on the master side return two bytes of status in addition to any data written by the slave. If any slave data is available, the status bytes are zero. If no data is present, the status bytes are the same as those returned by **PTY_INQUIRY**. |
| PTY_REMOTE_MODE | If this bit is set, data written by the master is sent as is to the slave side with no editing. |

| | |
|---|---|
| PTY_READ_WAIT | If this bit is set, a read on the master side is blocked until slave data is available. |
| PTY_HANDSHAKE_MODE | If this bit is set, a write on the master side is not complete until the slave consumes the data. |
| PTY_SLAVE_HOLD | If this bit is set, the slave is prohibited from writing any more data to the channel. |
| PTY_EOF | All slave accesses to the channel have been closed. |
| PTY_OUTPUT_QUEUED | The slave side has written data to the channel that has not yet been consumed by the master. |
| PTY_INPUT_QUEUED | The master has written data to the slave side that has not yet been consumed by the slave. |

*PTY_SET_MODE* is used to change the control mode for the pseudo-terminal channel. The value *<cval>* contains the new mode and should be some combination of the bits described in the previous section. The new control mode is exactly what is in *<cval>* so to perform an incremental change, the current value must be obtained using **PTY_INQUIRY**.

*PTY_FLUSH_READ* purges any data written by the master side to the slave input queue.

*PTY_FLUSH_WRITE* purges any data written by the slave side that has not yet been consumed by the master side.

*PTY_STOP_OUTPUT* prevents the slave side from writing any more data to the master side. This condition is reflected in the status bit *PTY_SLAVE_HOLD*.

*PTY_START_OUTPUT* allows the slave side to continue writing data to the master side.

## ERRORS REPORTED

| | |
|---|---|
| EIO | The file descriptor corresponds to slave mode access to the pseudo-terminal. |
| EINVAL | An argument to the function is invalid |

## NOTES

The file descriptor must correspond to master mode access to the pseudo-terminal.

## SEE ALSO

C Library: *create_pty(), read()*

## cos

Calculate the cosine of an angle.

## SYNOPSIS

```
#include <math.h>
double cos(r)
     double    r;
```

### Arguments

| | |
|---|---|
| <r> | The angle to use to compute the cosine |

### Returns

The cosine of the angle *<r>*

## DESCRIPTION

The **cos** function calculates the cosine of the angle *<r>*. **Cos** returns that value as its result.

The **cos** function interprets the value *<r>* as an angle expressed in radians, and returns a result between -1.0 and 1.0 inclusive.

## SEE ALSO

C Library: *acos(), sin(), tan()*

## cosh

Calculate the hyperbolic cosine of a value.

## SYNOPSIS

```
#include <math.h>
double cosh(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value to use to compute the hyperbolic cosine |

### Returns

The hyperbolic cosine of the argument *<x>*

## DESCRIPTION

The **cosh** function calculates the hyperbolic cosine of the value *<x>*. The hyperbolic cosine of *<x>* is defined as (exp(x) + exp(-x))/2. **Cosh** returns that value as its result.

The **cosh** function detects a range error if the magnitude of the hyperbolic cosine of *x* is larger than can be represented by the data type *double*. If **cosh** detects a range error, it calls **matherr()**, passing to it the address of a filled *struct* exception structure. **Cosh** sets the *<type>* element of the structure to **OVERFLOW**, *<name>* to the address of the character-string *<cosh>*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the function sets *<errno>* to **ERANGE**. The return value, which is system-dependent, is given in the tables in Section 3. If **matherr()** returns something other than 0, the function returns the value *retval* found in the *struct* exception structure as its result.

## SEE ALSO

C Library: *exp(), matherr()*

# creat

Create a new file or truncate an existing file.

## SYNOPSIS

```
#include <errno.h>
#include <sys/modes.h>
int creat(path, perms)
     char     *path;
     int       perms;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string that contains a pathname to the file you want to create or truncate |
| <perms> | A bit-string describing the access permissions to set on the created file |

### Returns

If successful, **creat** returns a file descriptor for the created or truncated file, otherwise **creat** returns **-1** with *<errno>* set to the system error code

## DESCRIPTION

If no file is reached by the pathname in the character-string referenced by the argument *<path>*, the **creat** function:

1. creates an empty file
2. assigns the current effective user-ID as the owner-ID of the file
3. assigns the access permissions to the file (described by anding the bit-string **<perms>** with the one's-complement of the current file-creation mask)
4. links the specified pathname to the file

The **creat** function then opens the file for writing access, ignoring the access permissions of the file, and sets the current file position to the beginning of the file.

If the pathname in the character-string referenced by *<path>* reaches a file, **creat** truncates the file to a length of zero and opens the file for writing access, setting the current file position to the beginning of the file. It does not change the access permissions or owner-ID of the file.

If **creat** succeeds, it returns a file descriptor for the opened file. Otherwise, **creat** returns **-1** with *<errno>* set to the system error code. **Creat** fails if it could not follow the path, the path contains a file that is not a directory, no more files can be created on the device to contain the file, or no more files can be opened by the task. **Creat** also fails if the pathname does not reach a file and the directory reached by the path does not grant the current effective user writing permission, or the pathname reaches a file that does not grant the current effective user writing permission.

*<Perms>* is a bit-string that describes the permissions to set on the file. The include-file *sys/modes.h* defines constants that describe the meanings of each bit used by **creat** in the bit-string. These constants are:

| | |
|---|---|
| S_IREAD | 0X01 |
| S_IWRITE | 0X02 |
| S_IEXEC | 0x04 |
| S_IOREAD | 0X08 |
| S_IOWRITE | 0X10 |
| S_IOEXEC | 0x20 |
| S_ISUID | 0X40 |

**S_IREAD** grants read permission to the owner of the file, **S_IWRITE** grants write permission to the owner, and **S_IEXEC** grants search permission to the owner if a directory, or **S_IEXEC** grants execution permission if a file. **S_IOREAD** grants read permission to users other than the owner of the file, **S_IOWRITE** grants write permission to others, and **S_IOEXEC** grants search permission to others if a directory, or **S_IOEXEC** grants execution permission if a file. **S_ISUID** causes the effective user-ID to change to the owner of the file when the program in the file is executed. The results of **creat** are undefined if bits other than those defined above are set in *<perms>*.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The existing file or the directory to contain the link to the new file does not grant the user writing permission |
| EMFILE | the maximum number of files are open |
| EMSDR | could not follow the path to this file |
| ENOSPC | The are no available file description nodes on the device that was to contain the specified file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The **creat** function opens the created file for writing, even if the access permissions assigned to the file do not grant write permission to the current effective user.

If the task has the maximum number of files open and the specified file does not exist, the **creat** function creates the file, but does not open it.

## SEE ALSO

C Library: *fcreat(), fopen()*

System Call: *chmod(), chown(), open(), umask()*

Command: **create**

# create_pty

Create a pseudo-terminal channel

## SYNOPSIS

```
#include <errno.h>
#include <sys/pty.h>
int create_pty(fds)
    int (*fds)[2]
```

### Arguments

| | |
|---|---|
| <fds> | A pointer to an array of two file descriptors |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **create_pty** function creates a new pseudo-terminal channel. The file descriptor for slave access is returned in *<fd[0]>*. The file descriptor for master access is returned in *<fd[1]>*.

Pseudo-terminals must exist as real devices in the device directory named */dev/pty00*, */dev/pty01*, etc. These devices may be created using the **makdev** utility using the command:

**makdev /dev/pty***xx* **p 1** *xx*

where *xx* is a decimal number with a possible leading zero.

The **create_pty** function returns access to the first unused pseudo-terminal channel in the system. As these channels are closed, they are reused in numerical order. That is, **create_pty** always returns the lowest numbered pseudo-terminal channel not currently in use.

Once the channel has been opened using **create_pty**, additional slave accesses may be obtained using **open** for the appropriate device.

For slave access, this channel is exactly the same as a normal terminal. For master access, writing to the channel is seen as input on the slave side and reading from the channel reads characters output from the slave side.

The **ofstat()** function may be applied to a pseudo-terminal. The only difference from a normal terminal is that the the mode is S_SLAVE_PTY or S_MASTER_PTY.

## ERRORS REPORTED

| | |
|---|---|
| ENOSPC | All pseudo-terminal devices are currently open. |
| EBADF | The device name **/dev/pty*xx* does not correspond to a pseudo-terminal.** |

## NOTES

None

## SEE ALSO

C Library: *control_pty(), mknod(), ofstat(), ttyname()*

# _crypt

Encrypt a character-string.

## SYNOPSIS

```
void _crypt(crypw, pw)
     char     *crypw;
     char     *pw;
```

### Arguments

| | |
|---|---|
| <crypw> | The address of the target buffer to get the encrypted string |
| <pw> | The address of the character-string to encrypt |

### Returns

Void

## DESCRIPTION

The **_crypt** function encrypts the first eight characters of the character-string (referenced by *<pw>*) using the standard encryption algorithm. This generates a character-string containing sixteen characters. **_Crypt** copies the generated character-string to the the target buffer referenced by *<crypw>*.

If *<pw>* references a null-string, it copies a null-string to the target buffer referenced by *<crypw>*.

## NOTES

The **_crypt** function produces unpredictable results if the length of the character-string is greater than eight characters.

## SEE ALSO

Command: **password**

# ctime

Generate an ASCII time stamp.

## SYNOPSIS

```
#include <time.h>
char *ctime(pclock)
     long     *pclock;
```

### Arguments

| | |
|---|---|
| <pclock> | The address of the system-time value |

### Returns

The address of the generated ASCII time stamp

## DESCRIPTION

The **ctime** function generates a time stamp from the system time value referenced by *<pclock>* that represents the date and time in the local time zone. **Ctime** returns the address of the generated time stamp.

A time stamp is a 26-character string (including the terminating null-character) consisting of:

- the day of the week
- the month of the year
- the day of the month
- the hour
- minute
- second
- year

The time stamp is generated by the **sprintf()** format:

```
"%3s %3s %2.2d %2.2d:%2.2d:%2.2d %4.4d\n"
```

## NOTES

The result of the **ctime** function is in static memory. Subsequent calls to **ctime()** or **asctime()** overwrite that memory.

The **ctime** function calls **tzset()**, which sets up **daylight**, **timezone**, and **tzname**.

A system time value is a *long* containing the number of seconds since the epoch. The epoch is 00:00 GMT (midnight) on January 1, 1980.

If the system time value referenced by <*pclock*> is less than <*timezone*>, **ctime()** generates a time stamp for 00:00 on January 1, 1980, locally.

## SEE ALSO

C Library: *asctime(), daylight, gmtime(), localtime(), sprintf(), timezone, tzname, tzset()*

System Call: *time*

Command: **date**

## daylight

Daylight savings time flag.

### SYNOPSIS

```
#include <time.h>
extern int daylight;
```

### DESCRIPTION

The *<daylight>* variable is non-zero if and only if United States standard daylight savings time is being observed. The *<daylight>* variable should be applied to all conversions of time expressed in the local time zone. Otherwise, *<daylight>* is zero.

The *<daylight>* variable is initialized automatically by **localtime()** and **ctime()** and may be started explicitly by **tzset()**. The value of the *<daylight>* variable is zero before initialization.

### SEE ALSO

C Library: *ctime(), localtime(), timezone, tzname, tzset()*

System Call: *ftime()*

## dirname

Extract the directory prefix from a pathname.

## SYNOPSIS

```
char *dirname(path)
     char      *path;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing the pathname |

### RETURNS

The address of a character-string containing the directory prefix

## DESCRIPTION

The **dirname** function extracts the directory prefix from the pathname in the character-string referenced by *<path>*. **Dirname** returns the address of a character-string containing the directory prefix.

If the pathname contains a '/' (slash) character, the **dirname** function copies all of the characters up to, but not including, the last '/' to the static result buffer of the function. If the result is a null-string, the **dirname** function copies "/" into the buffer. If the pathname contains no '/' character, the **dirname** function copies "." into the buffer. The result of this function references the static result buffer.

## NOTES

The character-string referenced by the result of the **dirname** function is in static memory and is overwritten by subsequent calls to this function.

The **dirname** function does not verify the pathname in the character-string referenced by *<path>*.

## SEE ALSO

C Library: *basename(), fullname()*

Command: **dirname**

## dup

Duplicate a file descriptor.

## SYNOPSIS

```
#include <errno.h>
int dup(fildes)
     int       fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor to duplicate |

### RETURNS

If successful, **dup** returns the duplicate file descriptor. Otherwise, **dup** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **dup** function duplicates the file descriptor *<fildes>*. The effect is of again opening the file referenced by *<fildes>*, using the same open-mode, and positioning to the current file position. If the **dup** function successfully duplicates the file descriptor *<fildes>*, **dup** returns the duplicate file descriptor. Otherwise, **dup** returns **-1** with *<errno>* set to the system error code.

The **dup** function fails if the task can not open any more files, the file descriptor is out of range, or the file descriptor does not reference an open file.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode. |
| EINVAL | An argument to the function is invalid. |
| EMFILE | The maximum number of files are open. |

## NOTES

The function always uses the lowest numbered available file descriptor.

## SEE ALSO

System Call: *close(), creat(), dup2(), open(), pipe()*

## dup2

Duplicate a file descriptor onto a specific file descriptor.

## SYNOPSIS

```
#include <errno.h>
int dup2(src, dest)
     int       src;
     int       dest;
```

### Arguments

| | |
|---|---|
| <src> | The file descriptor to duplicate |
| <dest> | The target file descriptor |

### RETURNS

If successful, **dup2** returns the duplicate file descriptor <*dest*>. Otherwise, **dup2** returns **-1** with <*errno*> set to the system error code.

## DESCRIPTION

If the file descriptor <*dest*> references an open file, the **dup2** function closes that file. **Dup2** then duplicates the file descriptor <*src*> onto the specified file descriptor <*dest*>. The effect is that of again opening the file referenced by <*src*>, using the same open-mode, and positioning to the current file position. If **dup2** successfully duplicates the file descriptor <*src*> onto the file descriptor <*dest*>, **dup2** returns the file descriptor <*dest*>. Otherwise, **dup2** returns **-1** with <*errno*> set to the system error code.

The **dup2** function fails if either of the file descriptors are out of range or the file descriptor <*src*> does not reference an open file.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor <*src*> does not reference an open file. |
| EINVAL | One or both of the file descriptors <*src*> and <*dest*> are out of range. |

## NOTES

If <*src*> and <*dest*> are the same file descriptor, **dup2** returns <*dest*> without checking either file descriptor for validity.

If <*dest*> references an open file, **dup2** does not close that file if the function fails.

## SEE ALSO

System Call: *close(), creat(), dup(), open(), pipe()*

# ecvt

Convert a floating-point value to a character-string.

## SYNOPSIS

```
char *ecvt(fp, count, pexp, psign)
     double    fp;
     int       count;
     int      *pexp;
     int      *psign;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to convert |
| <count> | The number of digits to produce |
| <pexp> | The address of the value to receive the decimal exponent |
| <psign> | The address of the value to receive the sign indicator |

### Returns

The address of the generated character-string.

## DESCRIPTION

The **ecvt** function breaks the floating-point value *<fp>* into a sign, a positive fractional part greater than or equal to 0.0 and less than 1.0, and a decimal exponent. **Ecvt** stores the decimal exponent through *<pexp>*. If the sign of *<fp>* is positive, **ecvt** stores 0 through *<psign>*, otherwise **ecvt** stores a non-zero value through *<psign>*. The **ecvt** function generates a character-string that contains the first *<count>* significant digits of the fractional part, with the last digit rounded, and returns as a result the address of that character-string.

## NOTES

If *<fp>* is not 0.0, the first digit of the character-string referenced by the result is not ´0´. Otherwise, **ecvt** generates a character-string containing *<count>* ´0´-characters, and it stores 0 through *<pexp>* and *<psign>*.

The **ecvt** function rounds the last digit generated, depending on what the next digit would have been had it been generated. **Ecvt** rounds up if the next digit would have been 5, 6, 7, 8, or 9. Otherwise, **ecvt** does not round up the last digit.

The character-string referenced by the result of **ecvt** is in static memory and is overwritten by subsequent calls to **ecvt**.

## SEE ALSO

C Library: *fcvt(), fprintf(), _ftoa(), gcvt()*

## edata

End-of-memory address of initialized data.

### SYNOPSIS

```
extern int edata;
```

### DESCRIPTION

The **edata** external label references the first byte of memory that is beyond (above) the last byte of the program's initialized data. There may or may not be addressable memory at that location. Since it is the value of the label that is interesting, this variable should only be referenced as

```
        &edata
```

### NOTES

The result of storing into **edata** is unpredictable.

### SEE ALSO

C Library: *end, etext*

## end

End-of-memory address of uninitialized data.

### SYNOPSIS

```
extern int end;
```

### DESCRIPTION

The **end** external label references the first byte of memory that is beyond (above) the last byte of the uninitialized data of the program before the program begins executing. The address of the first byte beyond the uninitialized data is also known as the *break address* of the program. (so, the value of **end** is the initial break address of the program). There may or may not be addressable memory at the location referenced by **end**. Since it is the value of the label that is interesting, this variable should only be referenced as

```
        &end
```

### NOTES

The result of storing into **end** is unpredictable.

The break address of the program can change during the execution of the program. The functions **malloc**(), **calloc**(), **realloc**(), and **free**() all have the potential of changing the break address, as do the system calls **sbrk**() and **brk**(). The function call

```
        sbrk(0)
```

returns the current break address without changing the current memory allocation of the program.

### SEE ALSO

C Library: *calloc(), edata, etext, free(), malloc()*

System Call: *brk(), sbrk()*

# endpwent

End password-file handling.

## SYNOPSIS

```
#include <pwd.h>
void endpwent();
```

### Arguments

None

### Returns

Void

## DESCRIPTION

The **endpwent** function ends password-file handling initiated by **getpwent()**, **getpwnam()**, or **getpwuid()**. **Endpwent** frees the resources allocated to those routines, and closes files opened by those routines. The **endpwent** function does nothing if **getpwent()**, **getpwnam()**, or **getpwuid()** has not been called, or **endpwent()** has been called since the last call to one of these functions.

## SEE ALSO

C Library: *getpwent(), getpwnam(), getpwuid(), setpwent()*

## errno

The system error code of the most recent system error.

### SYNOPSIS

```
extern int errno;
```

### DESCRIPTION

The **errno** external variable contains the error number of the most recent system error. Some C builtins also assign error codes to this variable.

### NOTES

The **errno** external variable is defined in the include-file *stdio.h*. C programs that include that file need not explicitly define this variable.

### SEE ALSO

C Library: *perror(), sys_errlist, sys_nerr*

# etext

End-of-memory address of program text.

## SYNOPSIS

```
extern int etext;
```

## DESCRIPTION

The **etext** external label references the first byte of memory that is beyond (above) the last byte of the text of the program. There may or may not be addressable memory at that location. Since it is the value of the label that is interesting, this variable should only be referenced as

```
        &etext
```

## NOTES

A text segment of a program contains the executable code and other constant data.

The result of storing into *etext* is unpredictable.

## SEE ALSO

C Library: *edata, end*

# execl

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execl(path, [arg0, [arg1, ...,[ argn,]]] nullp)
     char      *path;
     char      *arg0, *arg1, ..., argn;
     char      *nullp;
```

### Arguments

| | |
|---|---|
| <path> | The address of the character-string that contains a pathname of the executable file containing the program. |
| <arg0> | The address of the character-string that contains the argument zero to the new program (by convention this is the name of the new program). |
| <arg1> | The address of the character-string that contains argument one to the new program. |
| <argn> | The address of the character-string that contains the last argument to the new program. |
| <nullp> | A null-address (*(char *) NULL*) that ends the array of addresses of character-strings that contain arguments to the new program. |

### Returns

No returns if successful. Otherwise, **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **execl** function requests that:

- the operating system replace the program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by *<path>*
- that it pass as arguments to the new program the character-strings referenced by the values passed to this function following the argument *<path>* (through but not including the argument *<nullp>*), if any
- and that it begin executing the new program at its transfer address

When the new program begins, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the signal-handling mechanism of the new program like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by *<arg0>* through *<argn>*), and character-strings defining its environment (as defined by the environment list referenced by the external variable *<environ>*). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address *((char *) NULL)*.

The **execl** function only returns to the caller if the operating system reports an error. If the operating system reports an error, the **execl** function returns **-1** with *<errno>* set to the system error code.

**Execl** fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the file access permissions do not grant the current effective user execution permission. **Execl** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execl** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All C programs set themselves up so that the function **main()** has three parameters. The first parameter, defined as *int*, is the number of arguments passed to the program. The second parameter, defined as *char *[]*, is the address of the array of addresses, terminated by *((char *) NULL)*, which references character-strings containing the arguments to the program. The third parameter, defined as *char *[]*, is the address of the array of addresses, terminated by *((char *) NULL)*, which references character-strings containing the environmental information of the program. The external value *environ* is also set to this value.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execle(), execlp(), execv(), execve(), execvp(), fork(), profil(), signal(), vfork()*

Commands: **shell**

# execle

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execle(path, [arg0, [arg1, ...,[ argn,]]] nullp, envp)
     char     *path;
     char     *arg0, *arg1, ..., argn;
     char     *nullp;
     char     *envp[];
```

### Arguments

| | |
|---|---|
| <path> | The address of the character-string that contains a pathname for the file containing the program to execute. |
| <arg0> | The address of the character-string that contains the argument to the new program that is referenced as argument zero (by convention this is the name of the command). |
| <arg1> | The address of the character-string that contains argument one to the new program. |
| <argn> | The address of the character-string that contains the last argument to the new program. |
| <nullp> | A null-address (*(char *) NULL*) that ends the array of addresses of character-strings containing arguments to the new program. |
| <envp> | The address of an environment list defining the environment to pass to the new program. |

### Returns

Nothing is returned if **execle** is successful. Otherwise, **execle** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **execle** function requests that:

- the operating system replace the program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by *<path>*
- that **execle** passes as arguments to the new program the character-strings referenced by the values passed to this function following the argument *<path>* through but not including the argument *<nullp>*, if any,
- that **<execle>** passes as environment variables those found in the environment list referenced by *<envp>*
- and that **execle** begins executing the new program at its transfer address.

The argument *<envp>* references an environment list defining the environment variables to pass to the new program. An environment list is a variable length array of addresses of character-strings containing the definitions of the environment variables. This array is terminated by the null-address *((char *) NULL)*.

When the new program begins, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the signal-handling mechanism of the new program like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by *<arg0>* through *<argn>*), and character-strings defining its environment (as defined by the environment list referenced by *<envp>*). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address *((char *) NULL)*.

The **execle** function only returns to the caller if the operating system reports an error. If the operating system reports an error, **execle** returns **-1** with *<errno>* set to the system error code.

The **execle** function fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execle** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execle** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execl(), execlp(), execv(), execve(), execvp(), fork(), profil(), signal(), vfork()*

Commands: **shell**

## execlp

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execlp(file, [arg0, [arg1, ...,[ argn,]]] nullp)
     char     *path;
     char     *arg0, *arg1, ..., argn;
     char     *nullp;
```

### Arguments

| | |
|---|---|
| <file> | The address of the character-string containing the filename of the file containing the program to execute. |
| <arg0> | The address of the character-string that contains the argument to the new program, which is referenced as argument zero (by convention this is the name of the command). |
| <arg1> | The address of the character-string containing argument one to the new program. |
| <argn> | The address of the character-string containing the last argument to the new program. |
| <nullp> | A null-address (*(char *) NULL*) that ends the array of addresses of character-strings containing arguments to the new program. |

### Returns

Nothing is returned if **execlp** is successful. Otherwise, **execlp** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **execlp** function requests that:

- the operating system replace the program currently executing with the program found in the executable binary file (the new program) found by using the filename in the character-string referenced by *<file>*
- that it pass as arguments to the new program the character-strings referenced by the values passed to this function following the argument *file* through but not including the argument *<nullp>*, if any
- and that it begin executing the new program at its transfer address.

If the filename referenced by *<file>* contains a slash-character ('/'), the **execlp** function uses that name as the pathname to a file containing the program to execute. Otherwise, **execlp** follows the search rules specified by the *<path>* environment variable in the environment list referenced by the external variable *<environ>*.

When the new program begins, it inherits the following attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the signal-handling mechanism of the new program like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by *<arg0>* through *<argn>*), and character-strings defining its environment (as defined by the environment list referenced by the external variable *<environ>*). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address *((char *) NULL)*.

The **execlp** function only returns to the caller if the operating system reports an error. If the operating system reports an error, **execlp** returns **-1** with *<errno>* set to the system error code.

**Execlp** fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execlp** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execlp** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All C programs set themselves up so that the function **main**() has three parameters. The first parameter, defined as *int*, is the number of arguments passed to the program. The second parameter, defined as *char *[]*, is the address of the array of addresses, terminated by (*(char *) NULL*), which references character-strings containing the arguments to the program. The third parameter, defined as *char *[]*, is the address of the array of addresses, terminated by *((char *) NULL)*, which references character-strings containing the environmental information of the program. The external value <*environ*> is also set to this value.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execl(), execle(), execv(), execve(), execvp(), fork(), profil(), signal(), vfork()*

Commands: **setpath, shell**

## execv

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execv(path, argv)
     char     *path;
     char     *argv[];
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string that contains a pathname of the file that contains the program to execute |
| <argv> | The address of the argument list to pass to the new program |

### Returns

Nothing is returned if **execv** is successful. Otherwise, **execv** returns **-1** with *errno* set to the system error code.

## DESCRIPTION

The **execv** function requests that the operating system replace the program currently executing with the new program found in the executable binary file. The new program is reached by the pathname in the character-string referenced by *<path>*. The operating system should then pass as arguments to the new program those found in the argument list referenced by *<argv>*, and begin executing the new program at its transfer address.

The argument *<argv>* references an argument list that defines the arguments to pass to the new program. An argument list is a variable length array of addresses to character-strings containing the arguments to pass. The array of addresses is terminated by the null-address *((char *) NULL.*

When the new program begins, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID, unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID the owner-ID of the file. The operating system sets up the signal handling mechanism of the new program like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by the argument list referenced by <*argv*>), and character-strings defining its environment (as defined by the environment list referenced by the external variable <*environ*>). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address (*(char *) NULL*).

The **execv** function only returns to the caller if the operating system reports an error. If the operating system reports an error, **execv** returns **-1** with <*errno*> set to the system error code.

The **execv** function fails it could not follow the path, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execv** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execv** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All C programs set themselves up so that the function **main()** has three parameters. The first parameter, defined as *int*, is the number of arguments passed to the program. The second parameter, defined as *char *[]*, is the address of the array of addresses, terminated by (*(char *) NULL*), which references character-strings containing the arguments to the program. The third parameter, defined as *char *[]*, is the address of the array of addresses, terminated by (*(char *) NULL*), which references character-strings containing environmental information of the program. The external value *environ* is also set to this value.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execl(), execle(), execlp(), execve(), execvp(),fork(), profil(), signal(), vfork()*

Commands: **shell**

## execve

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execve(path, argv, envp)
     char      *path;
     char      *argv[];
     char      *envp[];
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string that contains a pathname of the file that contains the program to execute |
| <argv> | The address of the argument list to pass to the new program |
| <envp> | The address of an environment list defining the environment variables to pass to the new program |

### Returns

Nothing is returned if **execve** is successful. Otherwise, **execve** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **execve** function requests that the operating system replace the program currently executing with the new program found in the executable binary file. The new program is reached by the pathname in the character-string referenced by *<path>*. The operating system should then pass as arguments to the new program those found in the argument list referenced by *<argv>*, and begin executing the new program at its transfer address.

The argument *<argv>* references an argument list defining the arguments to pass to the new program. An argument list is a variable length array of addresses to character-strings containing the definitions of the environment variables to pass. The array of addresses is terminated by the null-address.

The argument *<envp>* references an environment list that defines the environment to pass to the new program. An environment list is a variable length array of addresses to character-strings containing the arguments to pass. The array of addresses is terminated by the null-address *((char *) NULL.*

When the new program begins, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID the owner-ID of the file. The operating system sets up the new program's signal handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by the argument list referenced by *<argv>*), and character-strings defining its environment (as defined by the environment list referenced by *<envp>*). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address *((char *) NULL)*.

The **execve** function only returns to the caller if the operating system reports an error. If **execve** reports an error, it returns **-1** with *<errno>* set to the system error code.

**Execve** fails if it could not follow the path, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execve** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execve** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All C programs set themselves up so that the function **main()** has three parameters. The first parameter, defined as *int*, is the number of arguments passed to the program. The second parameter, defined as *char *[]*, is the address of the list of addresses, terminated by *((char *) NULL)*, which references character-strings containing the arguments to the program. The third parameter, defined as *char *[]*, is the address of the list of addresses, terminated by *((char *) NULL)*, which references character-strings containing the program´s environmental information. The external value *<environ>* is also set to this value.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execl(), execle(), execlp(), execv(), execvp(), fork(), profil(), signal(), vfork()*

Commands: **shell**

## execvp

Execute a program found in an executable binary file.

## SYNOPSIS

```
#include <errno.h>
int execvp(file, argv)
     char     *file;
     char     *argv[];
```

### Arguments

| | |
|---|---|
| <file> | The address of a character-string that contains a pathname for the file containing the program to execute |
| <argv> | The address of an argument list defining the arguments to pass to the new program |

### Returns

Nothing is returned if **execvp** is successful. Otherwise, **execvp** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **execvp** function requests that the operating system replace the program currently executing with the new program found in the executable binary file. The new program is reached by using the filename the character-string referenced by *<file>*. The operating system should then pass as arguments to the new program those found in the argument list referenced by *<argv>*, and begin executing the new program at its transfer address.

If the filename referenced by *<file>* contains a slash-character ('/'), **execvp** uses that name as the pathname to a file containing the program to execute. Otherwise, **execvp** follows the search rules specified by the *PATH* environment variable in the environment list referenced by the external variable *<environ>*.

When the new program starts, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the signal-handling mechanism of the new program like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by *<arg0>* through *<argn>*), and character-strings defining its environment (as defined by the environment list referenced by the external variable *<environ>*). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address (*(char *) NULL*).

The **execvp** function only returns to the caller if the operating system reports an error. If **execvp** reports an error, it returns **-1** with *<errno>* set to the system error code.

**Execvp** fails if it could not follow the path, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execvp** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

The argument <*argv*> references an argument list defining the arguments to pass to the new program. An argument list is a variable length array of addresses to character-strings containing the arguments to pass. The array of addresses is terminated by the null-address (*(char *) NULL.*

When the new program begins, it inherits these attributes and resources from the calling program:

- task priority
- task-ID number
- parent task-ID number
- user-ID number
- controlling terminal number
- file-creation permissions-mask
- time remaining on an armed alarm-clock
- working directory
- all open files
- system and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set. If the set-user-ID mode-bit is set, the new program gets as its effective user-ID the owner-ID of the file. The operating system sets up the signal of the new program handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the stack of the new program so it contains the number of arguments to the new program, its argument list, its environment list, character-strings containing its arguments (as defined by the argument list referenced by <*argv*>), and character-strings defining its environment (as defined by the environment list referenced by the external variable <*environ*>). Both the argument list and the environment list are variable length arrays of addresses, terminated by the null-address (*(char *) NULL*).

The **execvp** function only returns to the caller if the operating system reports an error. If the operating system reports an error, **execvp** returns **-1** with <*errno*> set to the system error code.

The **execvp** function fails if it could not follow the path, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the access permissions of the file do not grant the current effective user execution permission. **Execvp** also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

## ERRORS REPORTED

| | |
|---|---|
| E2BIG | Too many arguments are specified. |
| EACCES | The permissions of the file do not grant the requested access type. |
| EBBIG | The executable file is too large. |
| EISDR | The file is a directory. |
| EMSDR | Could not follow the path to the file. |
| ENOENT | The pathname does not reach a file. |
| ENOEXEC | This file is not executable. |
| ENOTDIR | A part of the path is not a directory. |

## NOTES

The **execvp** function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All C programs set themselves up so that the function **main()** has three parameters. The first parameter, defined as *int*, is the number of arguments passed to the program. The second parameter, defined as *char *[]*, is the address of the array of addresses, terminated by *((char *) NULL)*, which references character-strings containing the arguments to the program. The third parameter, defined as *char *[]*, is the address of the array of addresses, terminated by *((char *) NULL)*, which references character-strings containing environmental information of the program. The external value *<environ>* is also set to this value.

Most C programmers observe the convention that the first argument to a program is the name of that program.

## SEE ALSO

C Library: *environ, system()*

System Call: *execl(), execle(), execlp(), execv(), execve(), fork(), profil(), signal(), vfork()*

Commands: **setenv, shell**

## exit

Exit the program.

## SYNOPSIS

```
void exit(code)
     int  code;
```

### Arguments

| | |
|---|---|
| <code> | The task-termination code |

### Returns

Void

## DESCRIPTION

The **exit** function flushes, then closes, all of the standard I/O streams currently open. Then, **exit** ends execution of the program by terminating the task, giving *<code>* to the operating system to use as the task-termination code.

## NOTES

**Exit** does not return to the caller.

To avoid flushing standard I/O streams, end the program using **_exit**().

## SEE ALSO

C Library: *_exit()*

System Call: *fork(), wait()*

## _exit

Exit the program.

## SYNOPSIS

```
void _exit(code)
     int  code;
```

### Arguments

| | |
|---|---|
| <code> | The task-termination code |

### Returns

Void

## DESCRIPTION

The **_exit** function ends execution of the program by terminating the task, and giving *<code>* to the operating system to use as the task-termination code.

## NOTES

The **_exit** function does not return to the caller.

The **_exit** function closes all of the files attached to standard I/O streams in the program. However, **_exit** does not flush data in buffered streams to the attached file prior to closing the file. To flush these buffers, exit the program using **exit()**.

## SEE ALSO

C Library: *exit()*

System Call: *fork(), wait()*

## exp

Calculate the exponential of a value.

## SYNOPSIS

```
#include <math.h>
double exp(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value to use to compute the exponential |

### Returns

The exponential of the argument *<x>*

## DESCRIPTION

The **exp** function calculates the exponential of the value *<x>*. The exponential of *<x>* is defined as e (2.718281828459...) raised to the *<x>* power. **Exp** returns the calculated value as its result.

**Exp** detects a range error if the exponential of *<x>* is larger than can be represented by the data type *double.* If **exp** detects a range error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. **Exp** sets the *<type>* element of the structure to **OVERFLOW**, *<name>* to the address of the character-string *<exp>*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the function sets *<errno>* to ERANGE. If **matherr()** returns something other than zero, **exp** returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *log(), matherr()*

## fabs

Absolute value function.

## SYNOPSIS

```
double fabs(x)
     double       x;
```

### Arguments

| | |
|---|---|
| <x> | Value whose absolute value to calculate |

### Returns

The absolute value of the argument <*x*>

## DESCRIPTION

The **fabs** function calculates the absolute value of the argument <*x*>. **Fabs** returns the calculated value as its result.

## fclose

Close a stream.

## SYNOPSIS

```
#include <stdio.h>
int fclose(stream)
    FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to close |

### Returns

Zero if successful, EOF otherwise

## DESCRIPTION

The **fclose** function closes the standard I/O stream *<stream>* and frees any resources which were automatically allocated to the stream. If the stream is opened for writing and is buffered, **fclose** flushes any buffered data to the associated file.

**Fclose** returns EOF if it encounters an error while closing the stream, otherwise it returns zero.

## SEE ALSO

C Library: *fdopen(), fflush(), freopen(), fopen(), stderr, stdin, stdout*

System Call: *close(), open()*

## fcntl

Control the behavior of a file

## SYNOPSIS

```
#include <errno.h>
#include <sys/fcntl.h>
int fcntl(fd, function)
    int fd;
    int function;
```

### Arguments

| | |
|---|---|
| <fd> | A file descriptor |
| <function> | A function code (described below) |

### Returns

The current state of the file, otherwise **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **fcntl** function is used to change or interrogate the behavior of a file in the system. Various behaviors may be modified on a file-by-file, task-by-task basis. Each behavior may be set/reset by using a specific function to the **fcntl()** function.

The function returns a mask that indicates the state of the modifyable behaviors.

Currently, the functions available are:

| | |
|---|---|
| FNOBLOCK | Subsequent read operations on this file descriptor do not cause the task to be suspended if no data is available. In this mode, the ENOINPUT error is returned if no data is available and the signal "INPUT READY" is sent to the task when data becomes available. |
| FBLOCK | Returns the file descriptor to normal blocking mode. |

The value returned is a combination of these state bits:

| | |
|---|---|
| F_NOBLOCK | Reads from the file will not cause the task to be suspended. Also, the INPUT READY signal is sent when input becomes available. |

## ERRORS REPORTED

| | |
|---|---|
| ENOINPUT | The file has been set for non-blocking reads and no data is available. |
| EBADF | The file descriptor does not reference an open file, or the file is not open in the proper mode. |
| EEXIST | The pathname already references a file |

## NOTES

The INPUT READY signal is only sent to a task after the appropriate file has been placed in NOBLOCK mode AND a read request from the file was unsuccessful because of insufficient data.

## SEE ALSO

C Library: *idfd()*

# fcvt

Convert a floating-point value to a character-string.

## SYNOPSIS

```
char *fcvt(fp, pos, pexp, psign)
     double    fp;
     int       pos;
     int      *pexp;
     int      *psign;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to convert |
| <pos> | The digit position to produce digits to |
| <pexp> | The address of the value to receive the decimal exponent |
| <psign> | The address of the value to receive the sign indicator |

### Returns

The address of the generated character-string

## DESCRIPTION

The **fcvt** function generates a character-string containing digits from the absolute value of *<fp>* until it generates the *<pos>* digit to the right of the decimal point if *<pos>* is positive, or the -*<pos>* digit to the left of the decimal point if *<pos>* is negative. The **fcvt** function rounds the last digit depending on the next digit that it would have generated. If the sign of *<fp>* is positive, **fcvt** stores 0 through *<psign>*, otherwise it stores a non-zero value through *<psign>*. The **fcvt** function determines what the decimal exponent is if the decimal point were placed in front of the generated digits and returns that value through *<pexp>*. The **fcvt** function returns the address of the generated character-string as its result.

## NOTES

The **fcvt** function generates a null-string if *<fp>* is 0.0 or its magnitude is not large enough to have any significant digits at or to the left of the digit position specified by *<pos>*.

The first character generated by **fcvt** is never a **0**.

**Fcvt** rounds the last digit depending on what the next digit would have been if the function had generated it. It rounds the last digit up if the next digit is **5, 6, 7, 8**, or **9**, otherwise it does not round up.

The character-string referenced by the result of **fcvt** is in static memory. Subsequent calls to **fcvt** overwrites this string.

## SEE ALSO

C Library: *ecvt(), fprintf(), _ftoa(), gcvt()*

## fdopen

Attach an open file to a stream.

## SYNOPSIS

```
#include <stdio.h>
FILE *fdopen(fildes, mode)
     int        fildes;
     char      *mode;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the file to attach |
| <mode> | The address of a character-string describing the requested open mode |

### Returns

The standard I/O stream where the open file is attached, or *(FILE *) NULL* if **fdopen** detected an error.

## DESCRIPTION

The **fdopen** function attaches the file referenced by the file descriptor *<fildes>* to a standard I/O stream. An open file descriptor is returned by the system-call functions **creat()**, **dup()**, **dup2()**, **open()**, and **pipe()**. Valid open modes are *r*, *w*, *a*, *r+*, *w+*, and *a+*, which stand for read, write and append with the *+* implying open for update (reading and writing). Read and write access begins at the current position in the file, and append access begins at the end of the file. The **fdopen** function is typically used to permit standard I/O functions on a file opened by some means other than the standard I/O function **fopen()**.

The **fdopen** function returns the standard I/O stream to which the file has been attached, or *(FILE *) NULL* if there is an error. Possible errors include a bad file descriptor *<fildes>*, an unknown open mode, or attempting to exceed the maximum open-stream limit.

## NOTES

The access mode is supposed to match the open mode of the file. This is not currently checked since there is no way to coax the open mode from the operating system given an open file number.

Files attached to streams using this routine should be closed using *fclose()* to ensure that the resources automatically allocated to the stream are released to the system and that any data gets flushed.

When a file is opened for update (reading and writing), both input and output may be performed on the resulting stream. An input operation may not be performed immediately following an output operation without an intervening *fseek*. An output operation may not be performed immediately following an input operation without an intervening *fseek* unless the input operation encounters an end of file condition.

When a file is opened for append (that is, open modes *a* or *a+*), it is impossible to overwrite information already in the file. When output is written to the stream, the current file pointer is disregarded and repositioned to the end of the file.

## SEE ALSO

C Library: *fclose(), freopen(), fopen(), fseek()*

System Call: *close(), dup(), dup2(), open(), pipe()*

# feof

Test the end-of-file indicator of a stream.

## SYNOPSIS

```
#include <stdio.h>
int feof(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream |

### Returns

Non-zero if the end-of-file indicator on the stream is set (on), zero otherwise

## DESCRIPTION

The **feof** function tests the end-of-file indicator on the standard I/O stream *<stream>*. The **feof** returns a non-zero value if the indicator is set, otherwise it returns zero.

A standard I/O function sets the end-of-file indicator of a stream when the function attempts to read data from the stream produce no data and no errors.

## NOTES

The **feof** function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once.

## SEE ALSO

C Library: *fdopen(), ferror(), fopen(), stderr, stdin, stdout*

# ferror

Test the error-indicator of a stream.

## SYNOPSIS

```
#include <stdio.h>
int ferror(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream |

### Returns

Non-zero if the error-indicator on the stream is set (on), zero otherwise

## DESCRIPTION

The **ferror** function tests the error-indicator on the standard I/O stream *<stream>*. **Ferror** returns a non-zero value if the indicator is set, otherwise it returns zero.

A standard I/O function sets the error-indicator of a stream if the function attempts to perform I/O on the stream and the operating system reports an error. The function **clearerr()** clears the error-indicator of a stream.

## NOTES

The **ferror** function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once.

## SEE ALSO

C Library: *clearerr(), feof(), fdopen(), fopen(), stderr, stdin, stdout*

# fflush

Flush a stream opened for write access.

## SYNOPSIS

```
#include <stdio.h>
int fflush(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to flush |

### Returns

Zero if successful, EOF otherwise

## DESCRIPTION

The **fflush** function flushes any buffered data written to the standard I/O stream <*stream*>. The stream must be opened for write or append access. The **fflush** function returns EOF if it encounters an error flushing the stream, otherwise it returns zero.

## SEE ALSO

C Library: *fclose(), fdopen(), freopen(), fopen(), stderr, stdin, stdout*

# fgetc

Read a character from a stream.

## SYNOPSIS

```
#include <stdio.h>
int fgetc(stream)
FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to read from |

### Returns

The character read if successful, otherwise EOF

## DESCRIPTION

The **fgetc** function reads the next character from the standard I/O stream *<stream>*. If **fgetc** succeeded, it returns that character as its result, cast into an *int* with no sign extension, otherwise it returns EOF.

## NOTES

The character read is considered to be an *unsigned char* so there is no sign extension when converting the character to an integer value for returning.

## SEE ALSO

C Library: *fdopen(), fopen(), fputc(), fread(), getc(), getchar(), stdin*

# fgets

Read a character-string from a stream.

## SYNOPSIS

```
#include <stdio.h>
char *fgets(ptr, count, stream)
     char     *ptr;
     int       count;
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the target buffer |
| <count> | The size of the target buffer |
| <stream> | The standard I/O stream to read from |

### Returns

The *<ptr>* argument if successful, *(char *) NULL* otherwise.

## DESCRIPTION

The **fgets** function reads characters from the standard I/O stream *<stream>* until it reads *<count>*-1 characters, it reads an end-of-line character, or it reaches the end of the file. **Fgets** writes these characters to the buffer with the address *<ptr>*. **Fgets** appends a null-character ('\0') onto the characters read, making a character-string, then returns *<ptr>* as its result.

If **Fgets** detects an error, it returns *(char *) NULL* and does not alter the target buffer.

## SEE ALSO

C Library: *fdopen(), fgetc(), fopen(), fputs(), gets(), stdin*

## fileno

Get a file descriptor for the file attached to a stream.

### SYNOPSIS

```
#include <stdio.h>
int fileno(stream)
     FILE     *stream;
```

#### Arguments

| | |
|---|---|
| <stream> | A standard I/O stream |

#### Returns

A file descriptor for the file attached to the stream

### DESCRIPTION

The **fileno** function returns a file descriptor for the file attached to the stream *<stream>*. This file descriptor can be used by various system-call functions, such as **read()**, and **write()**.

### NOTES

Results of **fileno** are undefined if *<stream>* does not reference an open stream.

The **fileno** function is implemented as a macro by the include-file *<stdio.h>*.

### SEE ALSO

C Library: *fdopen(), fopen(), stderr, stdin, stdout*

System Call: *dup(), dup2(), open(), read(), write()*

## finite

Determine if a double precision floating point number is not an infinity.

## SYNOPSIS

```
include <math.h>
     int finite(x)
         double x;
```

### Arguments

None

### Returns

The value to examine Non-zero if the value is not an infinity, zero otherwise.

## DESCRIPTION

This function determines if the value <*x*> is finite. Infinity is represented as an exponent of 2047 (maximum value), a zero fraction and a sign bit.

## NOTES

**Not-a-number** returns non-zero, so the function **isnan()** should be called before **finite()**.

## SEE ALSO

C.Library: *isnan(), matherr()*

## floor

Calculate the largest integer not greater than a value.

## SYNOPSIS

```
#include <math.h>
double floor(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The floating-point argument to the function |

### Returns

The largest integer not greater than *<x>*

## DESCRIPTION

The **floor** function calculates the largest integer that is not greater than the value *<x>*. It returns that value, represented as a *double*, as its result.

## SEE ALSO

C Library: *ceil()*

# fmod

Floating-point remainder function.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <x> | The dividend |
| <y> | The divisor |

### Returns

The remainder resulting from dividing *<x>* by *<y>*

## DESCRIPTION

The **fmod** function calculates the remainder resulting from the division of *<x>* by *<y>*. The remainder of *<x>* divided by *<y>* is defined as *<x>* if *<y>* is 0.0, otherwise some value *<z>* that has the same sign as *<x>* such that *<x>* = <i>*<y> + <z> for some integer value *<i>* and fabs(z) < fabs(y). **Fmod** returns the calculated value as its result.

## SEE ALSO

C Library: *fabs()*

# fopen

Open a file and attach it to a standard I/O stream.

## SYNOPSIS

```
#include <stdio.h>
FILE *fopen(pathnam, mode)
     char      *pathnam;
     char      *mode;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname to the file to open |
| <mode> | The address of a character-string containing the open mode |

### Returns

If **fopen** is successful, the stream where the open file is attached is returned, otherwise *(FILE *) NULL* returns.

## DESCRIPTION

The **fopen** function opens the file reached by the pathname in the character-string referenced by *<pathnam>*. The character-string referenced by *<mode>* describes to **fopen** the access type desired by the program. **Fopen** then attaches the open file to a standard I/O stream.

If **fopen** succeeds, it returns the standard I/O stream as its result. Otherwise, it returns *(FILE *) NULL*. The **fopen** function fails if the operating system reports an error, the program has the maximum number of streams open, or the open mode is not valid. If the operating system reports an error, *<errno>* contains the system error code.

The open mode describes the type of access requested for the file. Valid open modes are *r*, *w*, *a*, *r+*, *w+*, and *a+*, which stand for read, write and append with the *+* implying open for update (reading and writing).

If the open mode is *r*, **fopen** opens the file for reading. If the file already exists, it sets the current position at the beginning of the file. If *<pathnam>* does not reach a file, the function fails.

If the open mode is *w*, **fopen** opens the file for writing. If the file already exists, **fopen** truncates the file to a length of zero. Otherwise, **fopen** creates a file with a length of zero. **Fopen** sets the current position at the beginning of the file.

If the open mode is *a*, **fopen** opens the file for writing. If the file does not exist, **fopen** creates a file with a length of zero. **Fopen** sets the current position at the end of the file.

If the open mode is *r+*, **fopen** opens the file for reading and writing. If the file already exists, **fopen** sets the current position at the beginning of the file. If *<pathnam>* does not reach a file, **fopen** fails.

If the open mode is *w+*, **fopen** opens the file for reading and writing. If the file already exists, **fopen** truncates the file to a length of zero. Otherwise, **fopen** creates a file with a length of zero. **Fopen** sets the current position at the beginning of the file.

If the open mode is *a+*, **fopen** opens the file for reading and writing. If the file does not exist, **fopen** function creates a file with a length of zero. **Fopen** sets the current position at the end of the file.

## NOTES

The include-file *<stdio.h>* defines the data type *FILE*. This data type is a structure containing all of the information about an open stream.

For brevity, this and other manual pages discuss a pointer to the data type *FILE* as simply a *<stream>*, instead of calling it a pointer to a structure defining the characteristics of a stream.

When a file is opened for update (reading and writing), both input and output may be performed on the resulting stream. An input operation may not be performed immediately following an output operation without an intervening *fseek*. An output operation may not be performed immediately following an input operation without an intervening *fseek* unless the input operation encounters an end of file condition.

When a file is opened for append (that is, open modes *a* or *a+*) it is impossible to overwrite information already in the file. When output is written to the stream, the current file pointer is disregarded and repositioned to the end of the file.

## SEE ALSO

C Library: *close(), fdopen(), fgetc(), fgets(), fputc(), fputs(), fread(), freopen(), fseek(), fwrite()*

System Call: *close(), open()*

# fork

Create a new task.

## SYNOPSIS

```
#include <errno.h>
int fork()
```

### Arguments

None

### Returns

Nothing is returned if **fork** is successful, the child's task-ID to the parent (calling) task and zero to the child (created) task, otherwise **fork** returns **-1** with *<errno>* set to the system error code.

## DESCRIPTION

The **fork** function creates a new task (the child task) that is an exact copy of the current task (the parent task). If **fork** succeeds, it returns the child task's task-ID to the parent task and returns zero to the child task. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The child task is identical to the parent task in that it has the same task priority, user-ID, effective user-ID, controlling terminal information, file-creation permissions-mask, working directory, signal handling set-up, and profiling information.

The child task differs from the parent task in that its task-ID is different, its parent task-ID is the task-ID of the parent task, the data in its memory is an exact copy of that in the parent task's memory, its file descriptors are exact copies of those in the parent task, and its system and user CPU times are reset to zero.

A task-ID is a non-negative integer. Flushing or closing streams opened for write or append access at the **fork()** call may result in data being duplicated onto the file attached to the stream since buffers are copied to the child task by **fork()**.

## SEE ALSO

C Library: *exit(), _exit() execl(), execlp(), execv(), execvp(), vfork(), wait()*

## fprintf

Write formatted data to a stream.

## SYNOPSIS

```
#include <stdio.h>
int fprintf(stream, format [,arglist])
     FILE     *stream;
     char     *format;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to use to write formatted data |
| <format> | The address of a character-string containing a format description |

### Returns

The number of characters written to the stream, or EOF if an error occurred.

## DESCRIPTION

The **fprintf** function generates characters from the format description in the character-string referenced by *<format>* and the arguments in the argument-list *<arglist>*, if any, and writes these characters to the standard I/O stream *<stream>*. **Fprintf** returns as its result the number of characters written to the stream.

The format description in the character-string referenced by *<format>* contains literal characters and field descriptions. The **fprintf** function writes literal characters to the stream with no interpretation. The **fprintf** function interprets field descriptions to determine what characters it generates, what type of argument it consumes, if any, from the argument list *<arglist>*, and the type of conversion it performs. The number of arguments and the type of the arguments in the argument list *<arglist>* depends on the format description. The argument list can be omitted.

The field description describes to the **fprintf** function the various attributes of the field. The syntax of a field description is:

```
%[<flags>][<width>][.[<precn>]][<alt>]<type>
```

The % character introduces a field description. The <flags> part modifies slightly the function's definition of the different conversion types, and consists of *+*, *-*, and *#*. The <width> part describes the field's width, and is indicated by a string of decimal digits or the * (asterisk) character. The <precn> part describes the number of digits to produce from the associated argument. It must follow a . (period) character and is indicated by a string of decimal digits or an asterisk. The <alt> part contains the alternate data-size indicator, which is the *l* (letter ell) character. The <type> part is the type of the field, and is one of the characters in this string: *cdeEfgGosuxX%* . The <type> part ends a field description. The **fprintf** function lets the flags, the width, the precision, and the alternate data-size indicator to be omitted from the field description. **Fprintf** requires that a field description contain a type. Examples of valid field descriptions are **%d, %-+#7.4lx, %%**, and **%8d.**

The <flags> part of the field description contains flags which alter the function's interpretation the field as described by its other parts. The field description may contain any, all, or none of these flags:

+ This flag tells the function to generate a leading sign if the conversion is a signed conversion. Normally, the function omits the leading sign if the value is positive.

- This flag tells the function to pad the generated characters on the right if the field's width is larger than the number of characters generated. Normally, the function pads the generated characters on the left.

This flag tells the function to generate a leading space if the conversion is a signed conversion and the value is positive. The **fprintf** function ignores this flag if the field also contains the *+* flag.

# This flag tells **fprintf** to convert the value using an alternate conversion method. The individual field types define the alternate conversion methods.

The <width> part controls the field's width, which describes the minimum number of characters in the field. If **fprintf** generates fewer than that number of characters from the argument, it prefixes spaces to fill the field. If the field contains the '-' flag, it appends spaces to fill the field. If the width is indicated by the '*' character, **fprintf** treats the next argument in the argument list as an *int* and uses the value of that argument as the width of the field. **Fprintf** expands the width of the field if it generates more than that number of characters. If the field contains no width, **fprintf** sets the width to the number of characters it generates.

The <precn> part describes the precision of the field, which controls the number of characters generated from the argument. If the precision is indicated by the '*' character, **fprintf** treats the next argument in the argument list as an *int* and uses the value of that argument as the precision of the field. The **fprintf** function treats a null digit string in the precision specification as zero. If the field contains no precision, the value of the argument for that field determines the precision of the field.

The <alt> part contains the alternate data-size indicator, which tells **fprintf** to consume an argument that is not the standard size for the type of field.

The <type> part is the type of the field, which indicates the size of the argument expected, if any, and the kind of conversion performed, if any. This list describes the types, their function, and their interpretation of the field's flags, width, precision, and alternate data-size indicator:

c The **fprintf** function treats the next argument as an *int*. The **fprintf** function generates a character by casting that value into a *char* If the width is greater than 1, **fprintf** pads that character with blanks. The function ignores the precision, alternate data-size indicator, and the '+', ' ', and '#' flags on this type of field.

d The **fprintf** function treats the next argument as an *int* unless the field contains the alternate data-size indicator 'l', in which case the **fprintf** function treats the argument as a *long*. The **fprintf** function generates a string of decimal digits representing the absolute value of the argument. If the field contains an explicit precision and the function generates fewer digits than specified by the precision of the field, **fprintf** prefixes '0' characters until it generates the requested number of digits. If the value of the argument is negative, **fprinf** prefixes a '-' character to the digits. Otherwise, if the field contains the '+' flag, **fprinf** prefixes a '+' character, or if the field contains the ' ' flag, **fprinf** prefixes a ' ' character. If **fprinf** generates fewer characters than specified by the width of the field, it pads with blanks to fill the field. The **fprintf** function ignores the '#' flag on this type of field.

e The **fprintf** function treats the next argument as a *double*. The function generates a string of decimal digits containing a decimal point, representing the magnitude of the mantissa of the argument. The magnitude of the mantissa is always greater than or equal to 1.0 and less than 10.0, unless the value is 0.0, in which case the mantissa is 0.0. If the field contains an explicit precision, **fprintf** generates exactly that many digits to the right of the decimal point. If the explicit precision is zero, **fprintf** omits the decimal point unless the field contains the alternate method flag '#'. The **fprinf** function assumes a precision of 6 if the field does not contain an explicit precision. If the value of the argument is negative, **fprintf** prefixes a '-' character to the generated characters. Otherwise, if the field contains the '+' flag, it prefixes a '+' character, or if the field contains the ' ' flag, it prefixes a ' ' character. The **fprinf** function then generates a string of decimal digits containing at least two digits, representing the magnitude of the decimal exponent. If the exponent is negative, **fprinf** prefixes a '-' character to the digits, otherwise it prefixes a '+' character. It then appends an 'e' character to the mantissa string followed by the exponent string. If **fprintf** generates fewer characters than specified by the field's width, it pads with blanks to fill the field. The **fprinf** function ignores the alternate data-size indicator on this type of field.

E This type is exactly like the 'e' field type except that the **fprinf** function uses the character 'E' to introduce the exponent instead of the character 'e'.

f The **fprinf** function treats the next argument as a *double*. The **fprinf** function generates a string of decimal digits containing a decimal point that represents the magnitude of the argument. If the field contains an explicit precision, the function generates exactly that many digits to the right of the decimal point. If the explicit precision is zero, the **fprinf** function omits the decimal point unless the field contains the alternate method flag '#'. The **fprintf** function assumes a precision of 6 if the field does not contain an explicit precision. If the value of the argument is negative, the function prefixes a '-' character to the generated characters. Otherwise, if the field contains the '+' flag, **fprintf** prefixes a '+' character, or if the field contains the ' ' flag, **fprintf** prefixes a ' ' character. If **fprintf** generates fewer characters than specified by the field's width, **fprintf** pads with blanks to fill the field. The function ignores the alternate data-size indicator on this type of field.

g The **fprintf** function treats the next argument as a *double*. If the decimal exponent of the argument is less than -4, or greater than or equal to the specified precision, the **fprintf** function generates characters as described by the 'e' type, except that unless the field contains the alternate method flag '#', it omits all trailing zeros from the digits representing the mantissa and the decimal point if it omits all of the digits to the right of the decimal point. If the decimal exponent falls within that range **fprintf** generates characters as described by the 'f' type, with the same exception. The **fprintf** function ignores the alternate data-size indicator on this type of field.

G This type is exactly like the 'g' field type except that if the function generates an exponent, uses the character 'E' to introduce the exponent instead of the character 'e'.

o The **fprintf** function treats the next argument as an *unsigned int* unless the field contains the alternate data-size indicator 'l', in which case the function treats the argument as an *unsigned long*. The **fprintf** function generates a string of octal digits representing the value of the argument. If the field contains an explicit precision and the function generates fewer digits than specified by the field's precision, it prefixes '0' characters until it generates the requested number of digits. If the field contains the '#' flag, **fprintf** prefixes a '0' character to the digits. If **fprintf** generates fewer characters than specified by the field's width, it pads with blanks to fill the field. The **fprintf** function ignores the ' ' and '+' flags on this type of field.

s The **fprintf** function treats the next argument as a *char ** and assumes that the argument references a character-string (a string of characters terminated by a null-character). If the field contains a precision and the length of the character-string is greater than the precision, the function uses the number of characters specified by the precision. Otherwise, **fprintf** uses all of the characters from the string. If the number of characters used is less than the field's width, **fprintf** pads with blanks to fill the field. The function ignores the '+', ' ', and '#' flags and the alternate data-size indicator on this type of field.

u The **fprintf** function treats the next argument as an *unsigned int* unless the field contains the alternate data-size indicator 'l', in which case the function treats the argument as a *unsigned long*. The **fprintf** function generates a string of decimal digits representing the value of the argument. If the field contains an explicit precision and the function generates fewer digits than specified by the field's precision, it prefixes '0' characters until it generates the requested number of digits. If **fprintf** generates fewer characters than specified by the field's width, it pads with blanks to fill the field. The **fprintf** function ignores the '+', ' ', and '#' flags on this type of field.

x The **fprintf** function treats the next argument as an *unsigned int* unless the field contains the alternate data-size indicator 'l', in which case the function treats the argument as an *unsigned long*. The function generates a string of hexadecimal digits representing the value of the argument. The function uses in sequence the characters 'a', 'b', 'c', 'd', 'e', and 'f' to represent the hexadecimal digits larger than '9'. If the field contains an explicit precision and the function generates fewer digits than specified by the precision of the field, **fprintf** prefixes '0' characters until it generates the requested number of digits. If the field contains the '#' flag, **fprintf** prefixes the characters "0x" to the digits. If **fprintf** generates fewer characters than specified by the width of the field, it pads with blanks to fill the field. The **fprintf** function ignores the ' ' and '+' flags on this type of field.

X This type is exactly like the ′x′ field type except that the **fprintf** function uses in sequence the characters ′A′, ′B′, ′C′, ′D′, ′E′, and ′F′ to represent the hexadecimal digits larger than ′9′, and it prefixes the characters "0X" if the field contains the ′#′ flag.

% The **fprintf** function generates a single ′%′ character. The function ignores the field′s flags, width, precision, and alternate data-size indicator.

## NOTES

The **fprintf** function consumes the variable width and precision values in the argument list before determining the type of the field.

The **fprintf** function writes characters to the stream using **fputc()**. If the stream is buffered, standard I/O does not write characters to the file attached to the stream until it fills the stream′s buffer or closes the stream. If the stream is line-buffered (buffered and attached to a file that is a terminal), standard I/O does not write characters to the file attached to the stream until it fills the stream′s buffer, closes the stream, writes an end-of-line character (EOL) to the stream, or reads data from a terminal.

The results of the **fprintf** function are undefined if the number of arguments in the argument list <arglist> is less than the number required by the format description.

The **fprintf** function ignores the extra arguments in the argument list if that list contains more arguments than required by the format description.

The **fprintf** function produces undefined results from an incorrectly constructed field description.

The **fprintf** function produces no digits from a zero value for a field that contains an explicit precision of zero for the field types ′d′, ′o′, ′s′, ′u′, ′x′, and ′X′.

The **fprintf** function ignores the explicit width and precision of a field description if either is specified by the ′*′ character and the value obtained from the argument list is less than zero.

The include-file *<stdio.h>* defines the functions and constants available in standard I/O. This file must be included in the C source before the first reference to this function.

The C library contains two versions of this function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the function **pffinit()**. Otherwise, it loads the version which contains no floating-point conversions.

## SEE ALSO

C Library: *ecvt(), fcvt(), fdopen(), fopen(), fputc(), fscanf(), gcvt(), pffinit(), printf(), scanf(), sprintf(), sscanf(), stderr, stdout*

Command: **cc**

# fputc

Write a character to a stream.

## SYNOPSIS

```
#include <stdio.h>
int fputc(c, stream)
     char      c;
     FILE      *stream;
```

### Arguments

| | |
|---|---|
| <c> | The character to write |
| <stream> | The standard I/O stream to write to |

### Returns

The value written if successful, EOF otherwise.

## DESCRIPTION

The **fputc** function writes the character *<c>* to the standard I/O stream *<stream>*. **Fputc** returns the character written as its result if it successfully writes the character to the stream, otherwise, it returns EOF.

## NOTES

If the stream is buffered but not line-buffered, standard I/O does not write the character to the attached file until the buffer of the stream is full or the stream is closed.

If the stream is line-buffered, standard I/O does not write the character to the attached file until an end-of-line character (EOL) is written to the stream, a standard I/O function attempts to read data from a terminal, the stream´s buffer is full, or the stream is closed.

If the **fputc** function succeeds, it returns the value of the *char* argument *<c>* converted to *int* as though *<c>* were an *unsigned char.*

## SEE ALSO

C Library: *fdopen(), fgetc(), fopen(), fputs(), putc(), putchar()*

# fputs

Write a character-string to a stream.

## SYNOPSIS

```
#include <stdio.h>
int fputs(s, stream)
     char      *s;
     FILE      *stream;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to write to the stream |
| <stream> | The standard I/O stream to write to |

### Returns

Zero if successful, EOF otherwise

## DESCRIPTION

The **fputs** function writes the characters in character-string referenced by *<s>* to the standard I/O stream *<stream>*. The **fputs** function returns zero as its result if it successfully writes the characters to the stream, otherwise it returns EOF.

## NOTES

The **fputs** function does not write to the stream the null-character terminating the character-string.

If the stream is buffered but not line-buffered, standard I/O does not write the characters to the attached file until it fills the buffer of the stream or closes the stream.

If the stream is line-buffered, standard I/O does not write the character to the attached file until it writes an end-of-line character (EOL) to the stream, attempts to read data from a terminal, fills the buffer of the stream, or closes the stream.

## SEE ALSO

C Library: *fdopen(), fgets(), fopen(), fputc(), puts()*

## fread

Read data from a stream.

## SYNOPSIS

```
#include <stdio.h>
int fread(ptr, size, count, stream)
     char     *ptr;
     int       size;
     int       count;
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <ptr> | Address of the buffer to contain the data read |
| <size> | The size of an item to read |
| <count> | The maximum number of items to read |
| <stream> | The standard I/O stream |

### Returns

The number of complete items read, if any

## DESCRIPTION

The **fread** function reads at most *<count>* items of *<size>* bytes from the I/O stream *<stream>*, placing the data read into the buffer whose address is *<ptr>*. The **fread** function reads data until it reads the requested number of data items, reaches the end of the file, or detects an error on the input stream. The function returns the number of complete items read from the stream.

## NOTES

If the **fread** function reaches the end of the file or encounters an error while reading a data item, it writes that partial item to the target buffer but does not count that partially read item in the count of items read, which it returns as its result.

The target buffer needs no special boundary alignment.

If <*count*> is less than or equal to zero, the function does not attempt to read any data and returns zero as its result.

## SEE ALSO

C Library: *fdopen(), fopen(), fwrite()*

System Call: *read(), write()*

# free

Free a block of allocated memory.

## SYNOPSIS

```
void free(ptr)
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the block of memory to free |

### Returns

Void

## DESCRIPTION

The **free** function returns the block of memory with the address *<ptr>* to the arena of available memory. The block of memory must have been allocated by **malloc()**, **calloc()**, or **realloc()**.

## NOTES

If the argument *<ptr>* is the address of a block that has already been freed, or is some value other than one returned by **malloc()**, **calloc()**, or **realloc()**, the function corrupts the arena of available memory and makes subsequent calls to **malloc()**, **calloc()**, **realloc()**, and **free()** behave unpredictably.

## SEE ALSO

C Library: *calloc(), malloc(), realloc()*

System Call: *brk(), cdata(), sbrk()*

# freopen

Reopen an open stream.

## SYNOPSIS

```
#include <stdio.h>
FILE *freopen(pathnam, mode, stream)
     char      *pathnam;
     char      *mode;
     FILE      *stream;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname to the file to open and attach to the stream |
| <mode> | The address of a character-string containing the open mode |
| <stream> | The standard I/O stream to reopen |

### Returns

The argument *<stream>* if successful, *(FILE *) NULL* otherwise

## DESCRIPTION

The **freopen** function closes the standard I/O stream *<stream>*, opens the file reached by the pathname in the character string referenced by *<pathnam>*, with the open mode specified by the character-string referenced by *<mode>*, and attaches the newly opened file to the stream.

If **freopen** succeeds, it returns the standard I/O stream *<stream>* as its result. Otherwise, it returns *(FILE *) NULL.* The **freopen** function fails if the operating system reports an error, the stream is not open, the program has the maximum number of streams open, or the open mode is not valid. If the operating system reports an error, *<errno>* contains the system error code.

The open mode describes the type of access requested for the file. Valid open modes are *r, w, a, r+, w+*, and *a+*, which stand for read, write and append with the *+* implying open for update (reading and writing).

If the open mode is *r*, **freopen** opens the file for reading if the file already exists, setting the current position at the beginning of the file. If the pathname *<pathnam>* does not reach a file, **freopen** fails.

If the open mode is *w*, **freopen** opens the file for writing. If the file already exists, **freopen** truncates the file to a length of zero. Otherwise, **freopen** creates a file with a length of zero. The **freopen** function sets the current position at the beginning of the file.

If the open mode is *a*, **freopen** opens the file for writing. If the file does not exist, **freopen** creates a file with a length of zero. The **freopen** function sets the current position at the end of the file.

If the open mode is *r+*, **freopen** opens the file for reading and writing. If the file already exists, **freopen** sets the current position at the beginning of the file. If the pathname *pathnam* does not reach a file, **freopen** fails.

If the open mode is *w+*, **freopen** opens the file for reading and writing. If the file already exists, **freopen** truncates the file to a length of zero. Otherwise, it creates a file with a length of zero. The **freopen** function sets the current position at the beginning of the file.

If the open mode is *a+*, **freopen** opens the file for reading and writing. If the file does not exist, **freopen** creates a file with a length of zero. The **freopen** function sets the current position at the end of the file.

## NOTES

The **freopen** function is typically used to attach files to automatically opened streams, such as *stdin*, *stdout*, and *stderr*.

The file originally attached to <*stream*> is closed without regard to the eventual outcome of the function call.

When a file is opened for update (reading and writing), both input and output may be performed on the resulting stream. An input operation may not be performed immediately following an output operation without an intervening **fseek**. An output operation may not be performed immediately following an input operation without an intervening **fseek** unless the input operation encounters an end of file condition.

When a file is opened for append (that is, open modes *a* or *a+*) it is impossible to overwrite information already in the file. When output is written to the stream, the current file pointer is disregarded and repositioned to the end of the file.

## SEE ALSO

C Library: *fclose(), fdopen(), fopen(), fseek(), stderr, stdin, stdout*

System Call: *close(), open()*

## frexp

Separate the exponent from the mantissa of a floating-point value.

## SYNOPSIS

```
double frexp(fp, iptr)
     double    fp;
     int      *iptr;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to separate |
| <iptr> | The address of an integer to receive the exponent of the floating-point value. |

### Returns

The mantissa of the floating-point value <*fp*>

## DESCRIPTION

The **frexp** function splits the floating-point value <*fp*> into its mantissa and its exponent. It stores the exponent through <*iptr*> and returns the mantissa as its result.

All floating-point values are represented by a mantissa and an exponent. A non-zero value is represented by a mantissa with an absolute value is greater than or equal to 0.5 and less than 1.0 and an exponent that is a signed integer. The floating-point value represented by the mantissa and exponent is 2 raised to the power indicated by the exponent which is then multiplied by the mantissa. For example, the floating-point value 1.0 is represented by a mantissa of 0.5 and an exponent of 1. A floating-point 0.0 is represented by a mantissa of 0.0 and an exponent of 0.

## SEE ALSO

C Library: *ldexp(), modf()*

# fscanf

Read and interpret formatted data from a stream.

## SYNOPSIS

```
#include <stdio.h>
int fscanf(stream, format [, addrlist])
     FILE      *stream;
     char      *format;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to read from |
| <format> | The address of a character-string containing a format description |

### Returns

The number of items in the address-list *<addrlist>* that **fscanf** successfully assigns or EOF if an error occurs before it assigns any data.

## DESCRIPTION

The **fscanf** function reads and interprets data from the standard I/O stream *<stream<* according to the format description in the character-string referenced by *<format>*. Following the argument *<format>* in the argument list, **fscanf** expects a list of addresses of variables to receive the values it generates from the data it reads from the stream, if any. The **fscanf** function returns as its result the number of assignments it makes, or EOF if it encounters an error before making the first assignment.

The argument *<format>* is a character-string containing a format description, which describes the format of the data read from the stream. The format description consists of literal characters, white-space characters, and field descriptions, in any sequence.

Literal characters are all characters which are not white-space characters (as defined by *isspace()*), and not part of field descriptions. A literal character tells the function to match that character with the next character read from the stream. If it does not match exactly, **fscanf** ends.

White-space characters are the space (' '), end-of-line ('\n'), horizontal-tab ('\t'), form-feed ('\f'), and carriage-return ('\r') characters. A white-space character tells the function to read and consume characters from the input stream until it reaches a character which is not a white-space character or it reaches the end of the data. The next character available from the stream is the next character which is not a white-space character. The function does nothing with a white-space character in the format description if the next character from the stream is not a white-space character.

A field description tells **fscanf** how to interpret the next character or characters read from the stream. The field description tells the function the maximum number of characters to read, the form of the characters read, the type of value to assign any result to, and whether to perform an assignment. A field description has this syntax:

```
%[*][<width>][<flags>]<type>
```

The '%' character introduces the field description. The '*' character tells **fscanf** to suppress assigning the interpreted value to a variable. The <width> part tells **fscanf** the maximum number of characters to read to satisfy the field (excluding leading white-space characters, if the field type skips leading white-space characters). The <flags> part alters the type of assignment made by the function, and may be either the 'h' or the 'l' character. The <type> part defines the type of the field and may be any one of the characters in this string: **cdeEfFgGosux%[**.

The <type> part of the field description defines the type of the field. It indicates the expected form of the data and the data type to receive the interpreted characters, if any. This list describes the types, the expected form of the interpreted characters, and the expected data type:

c This field type tells **fscanf** to expect a single character or a field of characters. If the field has no explicit width <width>, the **fscanf** function consumes the next character of data. Unless the field contains the '*' flag, **fscanf** assigns the consumed character through the next address in the address list as though that address references a *char*. If the field has an explicit width, the **fscanf** function consumes at most <width> characters. Unless the field contains the '*' flag, **fscanf** assigns the consumed characters through the next address in the address list as though that address references an array of *char*. The *fscanf* function ignores the 'h' and 'l' flags in this type of field description.

d This field type tells **fscanf** to expect a field of decimal digits. The function skips leading spaces and permits a sign immediately preceding the decimal digits. It consumes characters as decimal digits, generating an integer, until it reaches a character which is not a decimal digit, reaches the end of the data, or it consumes the number of characters specified by the field's explicit maximum width <width>, if any. Unless the field contains the '*' flag, **fscanf** assigns the integer generated from the decimal digits to the value referenced by the next address in the address list. If the field contains the 'h' flag, it assigns the value as though the address references a *short*. If the field contains the 'l' flag, it assigns the value as though the address references a *long*. Otherwise, it assigns the value as though the address references an *int*.

e This field type tells **fscanf** to expect a field containing characters representing a floating-point value. The **fscanf** function skips leading spaces, then consumes an optionally signed string of decimal digits, possibly containing a decimal point, optionally followed by an exponent field, which contains an 'E' or an 'e' character, optionally followed by a sign, optionally followed by a string of decimal digits. The **fscanf** function assumes that the exponent is zero if the decimal digits are omitted. **Fscanf** consumes characters until is reaches an inappropriate character, reaches the end of the data, or it consumes the maximum number of characters specified by the field's maximum width <width>, if any. It generates a floating-point value by multiplying the value represented by the the first string of digits by 10 raised to the power as expressed by the exponent field. Unless the field contains the '*' flag, **fscanf** assigns the resulting floating-point value. If the field contains the 'l' flag, it assigns the value through the next address in the address list as though that address references a *double*. Otherwise, it assigns the value through the address as though it references a *float*. The **fscanf** function ignores the 'h' flag on this field type.

E This field type tells **fscanf** to expect a field containing characters representing a floating-point value. The function behaves as though the field were an 'e' field except that it assigns the value through the next address in the address list as though that address references a *double* unless the field contains the 'h' flag, in which case it assigns the value as though the address references a *float*. The **fscanf** function ignores the 'l' flag on this field type.

f This field type tells **fscanf** to expect a field containing characters representing a floating-point value. The **fscanf** function behaves as though the field were an 'e' type field.

F This field type tells **fscanf** to expect a field containing characters representing a floating-point value. The **fscanf** function behaves as though the field were an 'E' type field.

g This field type tells the **fscanf** function to expect a field containing characters representing a floating-point value. The **fscanf** function behaves as though the field were an 'e' type field.

G This field type tells the **fscanf** function to expect a field containing characters representing a floating-point value. The **fscanf** function behaves as though the field were an 'E' type field.

o This field type tells the **fscanf** function to expect a field of octal digits. Octal digits are the decimal digits '0' through '7' inclusive. The **fscanf** function skips leading spaces and permits a sign immediately preceding the octal digits. The **fscanf** function consumes characters as octal digits, generating an integer, until it reaches a character which is not a octal digit, reaches the end of the data, or it consumes the number of characters specified by the explicit maximum width <width> of this field, if any. Unless the field contains the '*' flag, **fscanf** assigns the integer generated from the octal digits to the value referenced by the next address in the address list. If the field contains the 'h' flag, it assigns the value as though the address references a *short*. If the field contains the 'l' flag, **fscanf** assigns the value as though the address references a *long*. Otherwise, it assigns the value as though the address references an *int*.

s This field type tells the **fscanf** function to expect a field of characters. The **fscanf** function skips leading spaces, then consumes characters until it reaches the end of the data, a space, or it consumes the number of characters specified by the maximum width <width> of the field, if any. Unless the field contains the '*' flag, it assigns the consumed characters, followed by a null-character, through the next address in the address list as though the address references an array of *char*. The **fscanf** function ignores 'h' and 'l' flags on this field type.

u This field type tells the **fscanf** function to expect a field of decimal digits. The **fscanf** function skips leading spaces then consumes characters as decimal digits, generating an integer, until it reaches a character which is not a decimal digit, the end of the data, or it consumes the number of characters specified by the field's explicit maximum width <width>, if any. Unless the field contains the '*' flag, **fscanf** assigns the integer generated from the decimal digits to the value referenced by the next address in the address list. If the field contains the 'h' flag, it assigns the value as though the address references an *unsigned short*. If the field contains the 'l' flag, it assigns the value as though the address references an *unsigned long*. Otherwise, it assigns the value as though the address references an *unsigned int*.

x This field type tells the **fscanf** function to expect a field of hexadecimal digits. Hexadecimal digits are the decimal digits with the addition of the characters *A, B, C, D, E,* and *F*, in order, representing the six hexadecimal digits greater than 9. The **fscanf** function accepts *a, b, c, d, e,* and *f* for their equivalent upper-case character. The **fscanf** function skips leading spaces and permits a sign immediately preceding the hexadecimal digits. It consumes characters as hexadecimal digits, generating an integer, until it reaches a character that is not a hexadecimal digit, reaches the end of the data, or consumes the number of characters specified by the explicit maximum width <width> of the field, if any. Unless the field contains the '*' flag, **fscanf** assigns the integer generated from the hexadecimal digits to the value referenced by the next address in the address list. If the field contains the 'h' flag, it assigns the value as though the address references a *short*. If the field contains the 'l' flag, it assigns the value as though the address references a *long*. Otherwise, it assigns the value as though the address references an *int*.

% This field type tells the **fscanf** function to expect a single '%' character. The function does not skip any leading spaces, it makes no assignment and it ignores the <width> part of the field along with the field's '*', 'h', and 'l' flags.

[ This field type tells the function to expect a field of characters consisting of a particular set of printable, ASCII characters. The set of characters, called a scanset, is defined by a scanset description that immediately follows the field description. The scanset description begins with the first character following the field description and ends with the next ']' character, unless that character immediately follows the field description or immediately follows a '^' character that immediately follows the field description, in which case the scanset description ends with next ']' character. (This is so the ']' character can explicitly be part of the scanset.)

The scanset contains all of the characters in the scanset description up to but not including the terminating ']' character, with two exceptions. The first exception is a '^' character beginning the description, which indicates that the scanset contains all of the printable ASCII characters that are not in the scanset description. The second exception is a '-' character preceded by a character that is lexicographically less than the character that follows the '-' character, in which case the function substitutes for the '-' character all of the characters between the preceding and following character. The **fscanf** function consumes characters until it reaches a character that is not in the scanset, reaches the end of the data, or it consumes the number of characters specified by the field's maximum width <width>, if any. Unless the field contains the '*' flag, it assigns the consumed characters, followed by a null-character, through the next address in the address list as though the address references an array of *char*. The function ignores 'h' and 'l' flags on this field type.

## NOTES

Data satisfies a field's description for *d, e, f, g, o, u,* and *x* field types if **fscanf** consumes at least one digit. The field type determines the list of acceptable digits.

Any character consumed satisfies a *c* type field.

Data satisfies a *s* type field if **fscanf** consumes at least one character that is not a space.

Data satisfies a *[* type field if **fscanf** consumes at least one character that fits in the scanset of the field.

The **fscanf** function treats an I/O error and an end-of-file error as though it reached the end of the data. Standard I/O sets the error flag on the stream if an error occurs and it sets the end-of-file flag if it reaches the end of the file.

There is no direct way to determine success or failure of **fscanf** of matching literal characters or of interpreting fields which do not produce an assignment.

Trailing white-space characters are not read by **fscanf** unless explicitly told to do so by the format description.

The most common mistake made when using **fscanf** is passing to the function the values of the variables to receive the results of this function, instead of the addresses of those variables.

The include-file *<stdio.h>* defines this function, other functions, macros, and constants used by standard I/O.

The C library contains two versions of this function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the function **pffinit()**. Otherwise, it loads the version which contains no floating-point conversions.

## SEE ALSO

C Library: *fdopen(), fopen(), fprintf(), fscanf(), isdigit(), isspace(), isxdigit(), pffinit(), printf(), scanf(), sprintf(), sscanf(), stdin, strtol()*

Command: **cc**

## fseek

Reposition a stream.

## SYNOPSIS

```
#include <stdio.h>
int fseek(stream, offset, type)
     FILE      *stream;
     long       offset;
     int        type;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to reposition |
| <offset> | A value indicating the desired position, in bytes |
| <type> | A value indicating type of positioning |

### Returns

Zero if the positioning was successful, EOF otherwise

## DESCRIPTION

The fseek function changes the current offset into the stream referenced by *<stream>*. If *<type>* is 0, the value *<offset>* is a byte offset from the beginning of the stream. If *<type>* is 1, the value *<offset>* is a byte offset from the current position in the stream. If *<type>* is 2, the value *<offset>* is a byte offset from the end of the stream.

The fseek function returns zero if it successfully repositioned the stream, otherwise it returns EOF.

## NOTES

If the **fseek** function is not successful, <*errno*> contains the error code.

A file may be extended by requesting a seek relative to the end of the file with a positive offset.

A file may not be positioned before its beginning.

Calling this function undoes any effect of **ungetc**().

A stream attached to terminal may not be repositioned.

## SEE ALSO

C Library: *fdopen(), fopen(), ftell(), rewind()*

System Call: *lseek()*

# fstat

Get the status of an open file.

## SYNOPSIS

```
#include <errno.h>
#include <sys/stat.h>
int fstat(fildes, bufad)
     int           fildes;
     struct stat   *bufad;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the open file to examine |
| <bufad> | The address of the structure to contain the status of the file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **fstat** function examines the file referenced by the file descriptor *<fildes>* and writes information describing the status of that file into the structure whose address is *<bufad>*. The function returns zero as its result if it successfully gets the status of the open file. Otherwise, **fstat** returns **-1** with *<errno>* set to the system error code.

The function fails if the file descriptor *<fildes>* is out of range or does not reference an open file.

The following structure is defined in the include-file *sys/stat.h* and defines the format of the data describing the status of the open file:

```
        struct stat
        {
             short      st_dev;
             short      st_ino;
             char       st_filler;
             char       st_mode;
             char       st_perm;
             char       st_nlink;
             short      st_uid;
             long       st_size;
             long       st_mtime;
             long       st_spr;
        };
```

The value *st_dev* is the device number of the device containing the file. *st_ino* is the file descriptor number (FDN) on the device describing the file. *st_filler* is an unused byte. *st_mode* is a bit-string describing the type of the file, described below. *st_perm* is a bit-string describing the permissions of the file, described below. *st_nlink* is the number of links to the file (as a single character field, this limits the link count for a file a maximum of 127). *st_uid* is the owner-ID of the file. *st_size* is the size of the file, in bytes. *st_mtime* is the last modification date and time for the file, in system-time. *st_spr* is unused.

These constants, defined in the include-files *sys/stat.h* and *sys/modes.h*, define the data in the bit-string *st_mode* that describe the type of file:

| | |
|---|---|
| S_IFMT | 0x4F |
| S_IFREG | 0x01 |
| S_IFBLK | 0x03 |
| S_IFCHR | 0x05 |
| S_IFPTY | 0x07 |
| S_IFDIR | 0x09 |
| S_IFPIPE | 0x41 |

The constant **S_IFMT** is a mask that when anded with the value *st_mode* yields the file type. After anding with the constant **S_IFMT**, *st_mode* produces **S_IFREG** if the file is a regular file, **S_IFBLK** if the file is a block-special file (block device), **S_IFCHR** if the file is a character-special file (character device), **S_IFPTY** if the file is a pseudo tty device, **S_IFDIR** if the file is a directory, or **S_IFPIPE** if the file is a pipe.

These constants, also defined in the include-files *sys/stat.h* and *sys/modes.h,* define the data in the bit-string *st_perm* that describe the access permissions of the file:

```
S_IREAD       0x01
S_IWRITE      0x02
S_IEXEC       0x04
S_IOREAD      0x08
S_IOWRITE     0x10
S_IOEXEC      0x20
S_ISUID       0x40
S_SLAVE_PTY   0x07
S_MASTER_PTY  0x87
```

**S_IREAD** grants reading permission to the owner of the file, **S_IWRITE** grants writing permission to the owner, and **S_IEXEC** grants searching permission to the owner if the file is a directory, otherwise it grants execution permission. **S_IOREAD** grants reading permission to users other than the owner of the file, **S_IOWRITE** grants writing permission to others, and **S_IOEXEC** grants searching permission to others if the file is a directory, otherwise it grants execution permission. **S_ISUID** causes the effective user-ID change to that of the owner of the file whenever the program contained in the file is executed. **S_SLAVE_PTY** indicates a pseudo tty device slave and **S_MASTER_PTY** indicates a pseudo tty device master.

## ERRORS REPORTED

| | |
|---|---|
| EDADF | The file descriptor does not reference an open file or the file is not open in the proper mode. |
| EINVAL | An argument to the function is invalid. |

## NOTES

The include-file *<sys/modes.h>* need not be included if the include-file *<sys/stat.h>* is included since *<sys/stat.h>* includes *<sys/modes.h>*.

## SEE ALSO

System Call: *creat(), dup(), dup2(), link(), open(), pipe(), stat(), utime()*

Command: **dir**

## ftell

Get the current position of a stream.

## SYNOPSIS

```
#include <stdio.h>
long ftell(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | A standard I/O stream |

### Returns

The current position of the stream, in bytes

## DESCRIPTION

The **ftell** function examines the standard I/O stream *<stream>*, determines its current position relative to the beginning of the stream, and returns a value indicating that position.

If the stream is opened for read access, the current position contains the next character to read. If the stream is opened for write or append access, the current position is where the next character is written.

## NOTES

The **ftell** function is not affected by a character pushed onto the stream by **ungetc()**.

The **ftell** function takes into account I/O buffering, which means it may return a different position than the system call **lseek()**.

## SEE ALSO

C Library: *fdopen(), fopen(), fseek(), rewind()*

System Call: *lseek()*

# ftime

Get the current time statistics for the operating system.

## SYNOPSIS

```
#include <sys/timeb.h>
int ftime(tbufaddr)
     struct timeb  *tbufaddr;
```

### Arguments

<tbufaddr> The address of a structure to get the current time information of the operating system

### Returns

Zero

## DESCRIPTION

The **ftime** function writes the current time statistics for the operating system into the structure whose address is *<tbufaddr>*. The function always returns zero as its result.

The following structure definition describes the data written to the structure whose address is *<tbufaddr>*:

```
struct timeb
{
     long      time;
     char      tm_tik;
     char      dstflag;
     short     timezone;
};
```

The value *<time>* is the current system-time. *<tm_tik>* is the number of ticks (hundredths of a second) that have passed since the last change in the system-time. *<dstflag>* is non-zero if converting to time coordinates of the local time zone requires the U. S. A. Standard Daylight Savings Time conversion, zero otherwise. If *timezone* is positive, it is the number of seconds the local time zone is west of Greenwich Mean Time (GMT), otherwise its absolute value is the number of minutes east of GMT. The include-file *sys/timeb* contains definitions defining this structure.

## ERRORS REPORTED

None

## NOTES

The system represents time in system-time, which is the number of seconds that has elapsed since the epoch. The system defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time (GMT).

## SEE ALSO

C Library: *gmtime(), localtime(), tzset()*

System Call: *stime(), time()*

Command: **date**

# _ftoa

Convert a floating-point value to a character-string.

## SYNOPSIS

```
char *_ftoa(fp)
     double    fp;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to convert |

### Returns

The address of the generated character-string

## DESCRIPTION

The **ftoa** function generates a character-string representing the floating-point value *<fp>*. **Ftoa** returns as its result the address of that character-string.

The function generates a character-string containing 21 characters. (This count does not include the null-character which terminates the character-string.) It generates a sign (+ or -) representing the sign of the value, followed by 14 decimal-digits with a decimal point following the first digit which represents the mantissa, followed by an *E* character, followed by a sign representing the sign of the exponent, followed by three decimal-digits representing the magnitude of the exponent.

## NOTES

The character-string referenced by the result of **ftoa** is in static memory and is overwritten by subsequent calls to this function.

## SEE ALSO

C Library: *atof(), fprintf(), sprintf()*

## ftw

Descend the specified directory structure.

## SYNOPSIS

```
#include <sys/stat.h>
#include <ftw.h>
int ftw(path,fun,num fd)
     char      *path;
     int       (*fun)();
     int       num_fd;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing the pathname of the directory in which to begin the directory descent. |
| <fun> | The address of a function called for each file in the directory descent. |
| <num_fd> | The maximum number of file descriptors to use during the directory descent. |

### Returns

Zero if the directory structure is exhausted, **-1** if it detects an error, or whatever non-zero value was returned by a call to *<fun>*.

## DESCRIPTION

This function recursively descends the directory structure starting at the directory *<path>*.

For each file in the structure, it calls the user specified function, *<fun>* passing to it, the address of a character-string containing the name of the current file, the address of a **struct stat** structure, containing information about the file, and an integer flag. The possible values of the flag are defined in the include-file **ftw.h** and are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be searched, and FTW_NS for a file on which a **stat** could not be performed.

The directory descent continues until all entries in the directory structure have been processed, the user defined function *<fun>* returns a non-zero value or some error is detected within **ftw**. If all entries have been processed, **ftw** returns zero. If *<fun>* returns a non-zero value, **ftw** terminates, returning whatever value *<fun>* returned. If **ftw** detects an error, it returns **-1**, with **errno** set to the correct error type.

The *<num_fd>* argument limits the number of file descriptors used but not the depth of the directory search. **Ftw** uses one file descriptor for each level in the directory structure. **Ftw** will consume as many file descriptors as it needs to perform the directory descent. It is therefore possible that **ftw** will consume all available file descriptors, leaving none for user functions. The user should estimate the maximum number of file descriptors he will need for his own purposes and limit **ftw** to a number which will ensure that this many will always be available. If **ftw** runs out of file descriptors, it will close upper levels of the directory structure to free file descriptors and reopen them when it needs to access the given level again.

## NOTES

Ftw does a **depth-first** search of the directory structure.

The argument <*num_fd*> should not be less than one and any number greater than the maximum number of open files per task will be taken as the maximum. The maximum number of open files per task is system dependent.

Ftw uses **malloc** to allocate storage during its operation.

## SEE ALSO

C Library: *malloc()*

System Call: *stat()*

## fullname

Generate the full pathname.

## SYNOPSIS

```
char *fullname(path)
     char     *path;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname |

### Returns

The address of a character-string containing the full pathname or
*(char *) NULL* if unsuccessful

## DESCRIPTION

The **fullname** function generates the fully-qualified pathname equivalent of the pathname in the character-string referenced by *<path>*. If **fullname** is successful, it returns the address of a character-string containing the fully-qualified pathname.

The function returns *(char *) NULL* if it could not determine the fully-qualified pathname equivalent of the pathname in the character-string referenced by *<path>*. **Fullname** fails if it could not determine the directory prefix in the given pathname or it could not change directories to the directory in the prefix.

## NOTES

The character-string referenced by the result of this function is in static memory and is overwritten by subsequent calls to **fullname**.

A fully-qualified pathname is the shortest pathname from the root directory of the root device of the filesystem.

## SEE ALSO

C Library: *basename(), dirname(), getcwd()*

# fwrite

Write data to a stream.

## SYNOPSIS

```
#include <stdio.h>
int fwrite(ptr, size, count, stream)
     char     *ptr;
     int       size;
     int       count;
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the buffer containing the data to write |
| <size> | The size of an item to write |
| <count> | The number of items to write |
| <stream> | The standard I/O stream to write data to |

### Returns

The number of complete items written, if any

## DESCRIPTION

The **fwrite** function writes *<count>* items of *<size>* bytes from the buffer whose address is *<ptr>* to the standard I/O stream *<stream>*. The function writes data until it writes the requested number of data items, or it detects an I/O error.

The function returns as its result the number of complete items written to the stream.

## NOTES

The data buffer whose address is *<ptr>* needs no special boundary alignment.

If *<count>* is less than or equal to zero, **fwrite** does not try to write any data and returns zero as its result.

If the stream is buffered, but not line-buffered, standard I/O does not write the character to the attached file until the stream´s buffer is full or the stream is closed.

If the stream is line-buffered, standard I/O does not write the character to the attached file until an end-of-line character (EOL) is written to the stream, a standard I/O function attempts to read data from a terminal, the stream´s buffer is full, or the stream is closed.

## SEE ALSO

C Library: *fdopen(), fopen(), fread()*

System Call: *read(), write()*

# gcvt

Convert a floating-point value to a character-string.

## SYNOPSIS

```
char *gcvt(fp, precn, buf)
     double    fp;
     int       precn;
     char     *buf;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to convert |
| <precn> | The maximum number of digits to produce |
| <buf> | The address of a buffer to contain the resulting character-string |

### Returns

The address of the buffer containing the resulting character-string if successful, or *(char *) NULL* otherwise.

## DESCRIPTION

The **gcvt** function generates a character-string representing the floating-point value *<fp>*. The format of the character-string it generates depends on the floating-point value and the requested maximum number of significant digits *<precn>*. The function places the resulting character-string in the array of *char* referenced by *<buf>*. If **gcvt** succeeds, it returns the address of the target buffer *<buf>*. If **gcvt** fails, it returns *(char *) NULL.*

The **gcvt** function fails if the requested maximum number of significant digits is less than or equal to 0 or is greater than the maximum number of significant digits permitted by the function **ecvt()**.

The format of the character-string generated depends on both the value of *<fp>* and the requested number of significant digits *precn*. If the decimal exponent of *<fp>* is less than -4, or greater than or equal to *precn*, **gcvt** generates the character-string using an ´e´-type format (described by **fprintf()**). If the decimal exponent is greater than or equal to -4 and is less than *<precn>*, **gcvt** generates the character-string using an ´f´-type format (also described by **fprintf()**). In both cases, **gcvt** removes all trailing zeros from the generated character-string, and it removes the decimal point if there are no digits following the decimal point.

## NOTES

The maximum precision permitted by **ecvt()** is always more than twice the decimal precision of the of the most accurate floating-point data type.

The **gvct** function rounds the last digit depending on what the next digit would have been if the function had generated it. It rounds the last digit up if the next digit is *5, 6, 7, 8*, or *9*, otherwise it does not round up.

## SEE ALSO

C Library: *ecvt(), fcvt(), fprintf(), _ftoa()*

## getc

Read a character from a stream.

## SYNOPSIS

```
#include <stdio.h>
int getc(stream)
     FILE      *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to read from |

### Returns

The character read if successful, otherwise EOF

## DESCRIPTION

The **getc** function reads the next character from the standard I/O stream *<stream>*. If **getc** succeeded, it returns that character as its result, cast into an *int* with no sign extension, otherwise it returns EOF.

## NOTES

The character read is considered to be an *unsigned char*, so there is no sign extension when converting the character to an integer value for returning.

The **getc** function is exactly like **fgetc**() and is included for compatibility with other systems.

## SEE ALSO

C Library: *fdopen(), fgetc(), fopen(), fputc(), fread(), getchar(), stdin*

## getchar

Read a character from the standard input stream.

## SYNOPSIS

```
#include <stdio.h>
int getchar()
```

### Arguments

None

### Returns

The character read if successful, otherwise EOF

## DESCRIPTION

The **getchar** function reads the next character from the standard I/O stream *<stdin>*. If **getchar** succeeded, it returns that character as its result, cast into an *int* with no sign extension, otherwise it returns EOF.

## NOTES

The character read is considered to be an *unsigned char* so there is no sign extension when converting the character to an integer value for returning.

## SEE ALSO

C Library: *fdopen(), fgetc(), fopen(), fputc(), fread(), getc(), stdin*

# getcwd

Get the pathname of the working directory.

## SYNOPSIS

```
char *getcwd(ptr, size)
     char     *ptr;
     int       size;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the buffer to receive the pathname of the working directory, or *(char *) NULL* |
| <size> | Size, in bytes, of the target buffer |

### Returns

The address of the character-string that contains the pathname to the working directory

## DESCRIPTION

The **getcwd** function generates a character-string containing the complete pathname of the working directory. If the length of that string is greater than *<size>*, **getcwd** returns *(char *) NULL*. If *<ptr>* is equal to *(char *) NULL*, **getcwd** allocates a buffer using **malloc()**, copies the generated character-string into the allocated buffer, and returns as its result the address of the allocated buffer. Otherwise, it copies the generated character-string into the buffer whose address is *<ptr>* and returns *<ptr>* as its result.

## NOTES

The **getcwd** function returns *(char *) NULL* if *<ptr>* is *(char *) NULL* and **malloc()** is unable to allocate *<size>* bytes of memory.

If *<ptr>* is *(char *) NULL*, the buffer allocated by **getcwd()** may be freed using **free()**.

## SEE ALSO

C Library: *malloc(), free()*

System Call: *chdir()*

Command: **path**

# getenv

Get information from the environment list.

## SYNOPSIS

```
char *getenv(ptr)
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the character-string containing the name to search for in the environment list |

### Returns

The address of a character-string containing the value in the environment list associated with the specified name, or ((**char** *) **NULL** if the name was not found in the list

## DESCRIPTION

This function searches the environment list for the variable-name found in the character-string referenced by *<name>*, and returns the address of a character-string containing the definition of that variable. If that variable is not defined in the environment list, the function returns the null-address ((**char** *) NULL).

## NOTES

The environment list is a variable-length array of addresses terminated by the null-address. Each address references a character-string defining a variable of the current environment. Each character-string is of the form *<name>=<value>* where *<name>* is the name of the environment variable and *<value>* is the definition of that variable.

The character-string referenced by the result of this function is the actual definition of the environment variable within the environment list. Altering this character-string alters the definition of the environment list.

## SEE ALSO

C Library: *environ, putenv()*

## get_FPU_control

Return the contents of the MC68881 control and status registers

## SYNOPSIS

```
#include <float_interrupt.h>
void get_FPU_control(buffer)
     struct FPU_control *buffer;
```

### Arguments

<buffer> The address of the structure to contain the contents of the MC68881 registers

### Returns

None

## DESCRIPTION

The **get_FPU_control** function returns the contents of the MC68881 control and status registers (FPCR and FPSR, respectively). The user may inspect and modify the contents of these registers.

This function expects *<buffer>* to be the address of a structure that is defined as:

```
struct FPU_control {
    struct control_register fpcr;   /* control register */
    struct status_register fpsr;    /* status register */
};
```

The organization of the individual registers is defined by these structures:

```
struct control_register {
    unsigned          :4;      /* unused */
    unsigned rnd      :2;      /* rounding mode */
    unsigned prec     :2;      /* rounding precision */
    unsigned inex1    :1;      /* inexact decimal input */
    unsigned inex2    :1;      /* inexact operation */
    unsigned dz       :1;      /* divide by zero */
    unsigned unfl     :1;      /* underflow */
    unsigned ovfl     :1;      /* overflow */
    unsigned operr    :1;      /* operand error */
    unsigned snan     :1;      /* signaling NAN */
    unsigned bsun     :1;      /* branch/set on unordered */
    unsigned          :16;     /* unused */
};

struct status_register {
    unsigned          :3;      /* unused */
    unsigned inex     :1;      /* accrued inexact */
    unsigned adz      :1;      /* accrued divide-by-zero */
    unsigned aunfl    :1;      /* accrued underflow */
    unsigned aovfl    :1;      /* accrued overflow */
    unsigned iop      :1;      /* invalid operation */
    unsigned inex1    :1;      /* inexact decimal input */
    unsigned inex2    :1;      /* inexact operation */
    unsigned dz       :1;      /* divide by zero */
    unsigned unfl     :1;      /* underflow */
    unsigned ovfl     :1;      /* overflow */
    unsigned operr    :1;      /* operand error */
    unsigned snan     :1;      /* signaling NAN */
    unsigned bsun     :1;      /* branch/set on unordered */
    unsigned quotient :7;      /* 7 least significant bits of quotient *
    unsigned s        :1;      /* sign of quotient */
    unsigned nan      :1;      /* not a number or unordered */
    unsigned i        :1;      /* infinity */
    unsigned z        :1;      /* zero */
    unsigned n        :1;      /* negative */
    unsigned          :4;      /* unused */
};
```

For a description of the fields in the control and status registers, refer to the MC68881 hardware manual.

## NOTES

The include-file <*float_interrupt.h*> contains the definitions of the above structures.

## SEE ALSO

C Library: *put_FPU_control()*

System Call: *FPU_resume(), get_FPU_exception(), put_FPU_exception()*

# get_FPU_exception

Access MC68881 coprocessor exception-information

## SYNOPSIS

```
#include <errno.h>
#include <float_interrupt.h>
int get_FPU_exception(buffer)
     struct FPU_interrupt_data *buffer;
```

### Arguments

| | |
|---|---|
| <buffer> | The address of a buffer which will receive the MC68881 coprocessor exception-information |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **get_FPU_exception** function accesses the exception information provided by the MC68881 coprocessor when it detects an error for which it is generating interrupts. This function is intended for use in an interrupt-handling routine when attempting to recover from errors detected by the MC68881 coprocessor.

The **get_FPU_exception** function returns zero if it successfully accesses the exception information, otherwise, it returns **-1** with *<errno>* set to the system error code.

The **get_FPU_exception** function expects *<buffer>* to be the address of a structure defined as:

```
        struct FPU_interrupt_data {
            struct state_frame FPU_frame;    /* FPU state frame */
            struct control_register fpcr;    /* control register */
            struct status_register fpsr;     /* status register */
            short *fpiar;                    /* instruction address register *
            fpreg fp[8];                     /* floating-point data registers
            long CPU_data_register[8];       /* CPU "D" registers */
            long CPU_address_register[8];    /* CPU "A" registers */
            struct exception_frame CPU_frame;   /* CPU exception frame */
        };
```

The individual components of this structure are defined as:

```
struct exception_frame {
    unsigned short sr;              /* CPU status register */
             short *CPU_pc;         /* CPU program counter */
             short frame_type;      /* exception frame type */
             short *pc;             /* program counter */
    unsigned short ir;              /* internal register */
    unsigned short operation;       /* operation word */
             short *address;        /* effective address */
};

struct control_register {
    unsigned        :4;     /* unused */
    unsigned rnd    :2;     /* rounding mode */
    unsigned prec   :2;     /* rounding precision */
    unsigned inex1  :1;     /* inexact decimal input */
    unsigned inex2  :1;     /* inexact operation */
    unsigned dz     :1;     /* divide by zero */
    unsigned unfl   :1;     /* underflow */
    unsigned ovfl   :1;     /* overflow */
    unsigned operr  :1;     /* operand error */
    unsigned snan   :1;     /* signaling NAN */
    unsigned bsun   :1;     /* branch/set on unordered */
    unsigned        :16;    /* unused */
};
```

```
struct status_register {
    unsigned          :3;      /* unused */
    unsigned inex     :1;      /* accrued inexact */
    unsigned adz      :1;      /* accrued divide-by-zero */
    unsigned aunfl    :1;      /* accrued underflow */
    unsigned aovfl    :1;      /* accrued overflow */
    unsigned iop      :1;      /* invalid operation */
    unsigned inex1    :1;      /* inexact decimal input */
    unsigned inex2    :1;      /* inexact operation */
    unsigned dz       :1;      /* divide by zero */
    unsigned unfl     :1;      /* underflow */
    unsigned ovfl     :1;      /* overflow */
    unsigned operr    :1;      /* operand error */
    unsigned snan     :1;      /* signaling NAN */
    unsigned bsun     :1;      /* branch/set on unordered */
    unsigned quotient :7;      /* 7 least significant bits of quotient *
    unsigned s        :1;      /* sign of quotient */
    unsigned nan      :1;      /* not a number or unordered */
    unsigned i        :1;      /* infinity */
    unsigned z        :1;      /* zero */
    unsigned n        :1;      /* negative */
    unsigned          :4;      /* unused */
};

typedef unsigned char fpreg[12]; /* one floating point register */

struct exception_frame {
    unsigned short sr;              /* CPU status register */
             short *CPU_pc;         /* CPU program counter */
             short frame_type;      /* exception frame type */
             short *pc;             /* program counter */
    unsigned short ir;              /* internal register */
    unsigned short operation;       /* operation word */
             short *address;        /* effective address */
};
```

The contents of these structures reflect the contents of the CPU and coprocessor registers at the time that the exception interrupt was generated. For interpretation of the information contained in these structures, consult the appropriate hardware manuals.

## ERRORS REPORTED

| | |
|---|---|
| EBDCL | The CPU cannot support the MC68881 coprocessor |
| ENOFPUDATA | There is no exception information to be accessed |

## NOTES

Exception information is available only after the MC68881 interrupts the CPU. The user must have previously enabled these interrupts by setting the appropriate bits in the MC68881 control register.

The user must call the routine **FPU_resume()** to resume execution of the interrupted MC68881 instruction and exit the interrupt-handling routine. Attempting to exit the interrupt-handling routine by using the *return* statement may lead to unpredictable results because the program counter stored on the stack may not be correct.

The include-file *<float_interrupt.h>* contains the above structure definitions.

## SEE ALSO

C Library: *get_FPU_control(), put_FPU_control()*

System Calls: *FPU_resume(), put_FPU_exception()*

# geteuid

Get the effective user-ID number of the current task.

## SYNOPSIS

```
int geteuid()
```

### Arguments

None

### Returns

The effective user-ID number of the current task

## DESCRIPTION

The **geteuid** function gets the effective user-ID number of the current task and returns that value as its result.

## ERRORS REPORTED

None

## SEE ALSO

System Call: *getuid(), setuid()*

# getpass

Get a password using a prompt.

## SYNOPSIS

```
char *getpass(prompt)
     char     *prompt;
```

### Arguments

| | |
|---|---|
| <prompt> | The address of the character-string containing the prompt |

### Returns

The address of a character-string containing the password read, or *(char *) NULL* if there was an error

## DESCRIPTION

The **getpass** function writes the characters in the character-string referenced by *<prompt>* to the standard I/O output stream *<stderr>*. **Getpass** clears the echo attribute on the terminal associated with the standard I/O input stream *<stdin>*, then reads characters from *<stdin>* up to the first end-of-line character (EOL) or to the end of the file, saving the first eight characters in a static buffer, discarding the remaining characters if any and the end-of-line character, if any.

**Getpass** restores the echo attribute on the terminal to its original state, terminates the characters saved with a null-character, completing the character-string, then returns the address of that character-string as its result.

If **getpass** encounters an error, it restores the terminal to its original state and returns *(char *) NULL* as its result.

## NOTES

The **getpass** function uses standard I/O and may enlarge a program more than expected.

Nothing is written to *<stderr>* if *<prompt>* is *(char *) NULL.*

The **getpass** function catches keyboard, quit, alarm, and hang-up signals. If it catches a signal, it resets the terminal to its original configuration then resignals the signal so the calling program can handle that signal.

If **getpass** returns indicating an error, *<errno>* contains the system error code.

The character-string referenced by the result of **getpass** is in static memory and is overwritten by subsequent **getpass** calls.

The standard I/O output stream *<stderr>* must be attached to a terminal unless *<prompt>* is *(char *) NULL.* Otherwise, **getpass** returns *(char *) NULL* with *<errno>* set to ENOTTY.

The standard I/O input stream *<stdin>* must be attached to a terminal or **getpass** returns *(char *) NULL* with *<errno>* set to ENOTTY.

## SEE ALSO

C Library: *fputs(), gets(), stderr, stdin*

## getpid

Get task-ID number of the current task.

## SYNOPSIS

```
int getpid()
```

### Arguments

None

### Returns

The task-ID number of the current task

## DESCRIPTION

The **getpid** function gets the task-ID number of the current task and returns that value as its result.

## ERRORS REPORTED

None

## SEE ALSO

System Call: *exec(), fork()*

Command: **status**

## getppid

Get the task-ID number of the parent of the current task.

## SYNOPSIS

```
int getppid()
```

### Arguments

None

### Returns

The task-ID number of the parent of the current task

## DESCRIPTION

This function gets the task-ID number of the parent of the current task and returns that value as its result.

## ERRORS REPORTED

## SEE ALSO

System Call: *exec(), fork(), getpid()*

Command: *status*

# getpw

Get a password-file entry based on a user-ID.

## SYNOPSIS

```
int getpw(uid, ptr)
     int       uid;
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <uid> | The user-ID number to search for |
| <ptr> | The buffer to contain the record found |

### Returns

Zero if a record was successfully found, EOF otherwise

## DESCRIPTION

The **getpw** function searches the password-file of the system for the first correctly formatted record with a user-ID field equivalent to *<uid>*. If one is found, it copies that record, including the terminating end-of-line character (EOL), into the buffer with the address *<ptr>* and returns zero as its result. Otherwise, **getpw** leaves the buffer whose address is *<ptr>* unchanged and returns EOF as its result.

## NOTES

The **getpw** function is obsolete but is included for compatibility with older systems. New applications should use **getpwuid()**.

The caller is responsible for ensuring that the buffer with the address *<ptr>* is large enough to hold the data.

The **getpw** function uses standard I/O and may make the calling program larger than expected.

The password file of the system is */etc/log/password.*

## SEE ALSO

C Library: *getpwent(), getpwuid()*

Command: **password**

# getpwent

Get and decode the next entry in the system password file.

## SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwent();
```

### Arguments

None

### Returns

The address of the structure containing information from the record read, or *(struct passwd *) NULL* if no record was read

## DESCRIPTION

The **getpwent** function reads and decodes the next correctly formatted entry in the system password file. The information is saved in a static structure (defined below) and the address of that structure is returned as the its result. If **getpwent** could not read a record from the system password file, it returns *(struct passwd *) NULL* as its result.

If no previous **getpwent()**, **getpwnam()**, or **getpwuid()** has been successfully attempted, or **endpwent()** has been called since the last call to **getpwent()**, **getpwnam()**, or **getpwuid()** this function opens the system password file and positions it to the first record in the file. After **getpwent** function completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The **endpwent()** function closes the system password file. Task termination also closes the file. The function **setpwent()** rewinds the system password file, positioning it to the first record of the file.

The include-file *<pwd.h>* defines structures and constants used when manipulating the data in the system password file. The format of the *struct* passwd structure referenced by the result of this function is:

```
struct passwd
{
      char      *pw_name;
      char      *pw_passwd;
      int        pw_uid;
      char      *pw_dir;
      char      *pw_shell;
};
```

The entry *pw_name* is the address of a character-string containing the user-name. *pw_passwd* is the address of a character-string containing the encrypted password. *pw_uid* contains the user's identifying number (user-ID). *pw_dir* is the address of a character-string containing the user's home directory. *pw_shell* is the address of a character-string containing the shell-command for the first program to run after logging on. An encrypted password address of *(char *) NULL* indicates there is no password. A shell-command address of *(char *) NULL* indicates the initial program is the standard shell.

## NOTES

The structure referenced by the result of this function and the character-strings referenced by the values in that structure are in static memory and are overwritten by subsequent calls to **getpwent()**, **getpwnam()**, and **getpwuid()**.

The **getpwent** function ignores improperly formatted records in the system password file.

The **getpwent** function uses standard I/O and enlarges more than expected a program not otherwise using standard I/O.

The **getpwent** function returns *(struct passwd *) NULL* if the user does not have permission to access the password file, the current position on the system password file is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another.

The system password file is */etc/log/password.*

## SEE ALSO

C Library: *endpwent(), getpw(), getpwnam(), getpwuid(), putpwent(), setpwent()*

Command: **password**

# getpwnam

Get and decode the next entry in the system password file containing the given user-name.

## SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwnam(name)
     char     *name;
```

### Arguments

| | |
|---|---|
| <name> | The address of a character-string containing the user-name |

### Returns

The address of the structure containing the information in the record read, or *(struct passwd *) NULL* if no record was read

## DESCRIPTION

The **getpwnam** function reads and decodes the next correctly formatted entry in the system password file that contains a user-name matching that in the character-string referenced by the argument *<name>*. The information is saved in a static structure (defined below) and the address of that structure is returned as the its result. If **getpwnam** could not find a record in the system password file containing the specified user-name, **getpwnam** returns *(struct passwd *) NULL* as its result.

If no previous **getpwent()**, **getpwnam()**, or **getpwuid()** has been successfully attempted, or **endpwent()** has been called since the last call to **getpwent()**, **getpwnam()**, or **getpwuid()**, the **getpwnam** function opens the system password file and positions it to the first record in the file. After the **getpwnam** function completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The **endpwent()** function closes the system password file. Termination of the task also closes the password file. The **setpwent()** function rewinds the system password file.

The include-file *<pwd.h>* defines constants and structures used when manipulating entries in the system password file. The format of the *struct* passwd structure referenced by the result of this function is:

```
struct passwd
{
     char     *pw_name;
     char     *pw_passwd;
     int       pw_uid;
     char     *pw_dir;
     char     *pw_shell;
};
```

The *pw_name* entry is the address of a character-string containing the user-name. *pw_passwd* is the address of a character-string containing the encrypted password. *pw_uid* contains the user's identifying number (user-ID), *pw_dir* is the address of a character-string containing the user's initial home-directory, and *pw_shell* is the address of a character-string containing the shell-command for the first program to run after logging on. An encrypted password address of *(char *) NULL* indicates that there is no password. A shell-command address of *(char *) NULL* indicates that the initial program is the standard shell.

## NOTES

The structure referenced by the result of the **getpwnam** function and the character-strings referenced by the values in that structure are in static memory and are overwritten by subsequent calls to **getpwent()**, **getpwnam()**, and **getpwuid()**.

The **getpwnam** function ignores improperly formatted records.

The **getpwnam** function uses standard I/O and enlarges more than expected a program not otherwise using standard I/O. The **getpwnam** function returns *(struct passwd *) NULL* if permissions deny access the password file, the current position on the system password file is end-of-file, or there is the maximum number of standard I/O streams open.

The system password file is */etc/log/password.*

## SEE ALSO

C Library: *endpwent(), getpw(), getpwent(), getpwuid(), putpwent(), setpwent()*

Command: **password**

# getpwuid

Get and decode the next entry in the system password file containing the given user-ID number.

## SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwuid(uid)
     int       uid;
```

### Arguments

| | |
|---|---|
| <uid> | The user-ID number to search for |

### Returns

The address of the structure containing the information in the record read, or *(struct passwd *) NULL)* if no record was read

## DESCRIPTION

The **getpwuid** function reads and decodes the next correctly formatted entry in the system password file that contains a user-ID number matching the user-ID number *<uid>*. The information is saved in a static structure (defined later) and the address of that structure is returned as the result. If **getpwuid** could not find a record in the system password file containing the specified user-ID number, **getpwuid** returns *(struct passwd *) NULL* as its result.

If no previous **getpwent()**, **getpwnam()**, or **getpwuid()** has been successfully attempted, or **endpwent()** has been called since the last call to **getpwent()**, **getpwnam()**, or **getpwuid()**, the **getpwuid** function opens the system password file and positions it to the first record in the file. After **getpwuid** completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The **endpwent()** function closes the system password file. Task termination also closes the file. The **setpwent()** function rewinds the system password file, positioning it to the beginning of the first record in the file.

The include-file *<pwd.h>* defines structures and constants used when reading and manipulating entries in the system password file. The format of the *struct* passwd structure referenced by the result of this function is:

```
struct passwd
{
     char      *pw_name;
     char      *pw_passwd;
     int        pw_uid;
     char      *pw_dir;
     char      *pw_shell;
};
```

The *pw_name* entry is the address of a character-string containing the user-name. *pw_passwd* is the address of a character-string containing the encrypted password. *pw_uid* contains the user's identifying number (user-ID). *pw_dir* is the address of a character-string containing the user's initial home-directory. *pw_shell* is the address a character-string containing the shell-command for the first program to run after logging on. An encrypted password address of *(char *) NULL* indicates there is no password. A shell-command address of *(char *) NULL* indicates that the initial program is the standard shell.

## NOTES

The structure referenced by the result of **getpwuid** and the character-strings referenced by the values in that structure are in static memory and are overwritten by subsequent calls to **getpwent()**, **getpwnam()**, and **getpwuid()**.

Improperly formatted records in the system password file are ignored.

The **getpwuid** function uses standard I/O and enlarges more than expected a program not otherwise using standard I/O.

The **getpwuid** function returns *(struct passwd *) NULL* if the user does not have permission to access the password file, the current position on the system password file is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another.

The system password file is */etc/log/password.*

## SEE ALSO

C Library: *endpwent(), getpw(), getpwent(), getpwnam(), putpwent(), setpwent()*

Command: **password**

# gets

Read a character-string from the standard input stream.

## SYNOPSIS

```
#include <stdio.h>
char *gets(ptr)
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the target buffer |

### Returns

The *<ptr>* argument if successful, *(char *) NULL* otherwise

## DESCRIPTION

The gets function reads characters from the standard I/O input stream *<stdin>* until it reads an end-of-line character, or reaches the end of the file. It places the characters in the buffer with the address *<ptr>*. If the last character read was an end-of-line character, **gets** replaces that character with a null-character, otherwise, it appends a null-character onto the characters read, making a character-string.

If gets is successful, meaning it read at least one character, it returns *<ptr>* as its result, otherwise. it returns *(char *) NULL* as its result and does not alter the target buffer.

## SEE ALSO

C Library: *fdopen(), fgets(), getc(), puts(), stdin*

# getuid

Get the user-ID number of the current task.

## SYNOPSIS

```
int getuid()
```

### Arguments

None

### Returns

The user-ID number of the current task

## DESCRIPTION

The **getuid** function gets the user-ID number of the current task and returns that value.

## ERRORS REPORTED

None

## SEE ALSO

System Call: *geteuid(), setuid()*

# getw

Read a word from a standard I/O stream.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to read from |

### Returns

The value read if successful, EOF otherwise

## DESCRIPTION

The getw function reads the next *sizeof(int)* bytes from the stream *<stream>*, assigns them to an *int* and returns that value as its result. If **getw** detects an error or reaches the end of the stream, it returns EOF.

## NOTES

The value EOF is a valid value to read, so the functions **ferror()** and **feof()** should be used to check for error and end-of-file conditions on the stream.

The **getw** function ignores odd bytes at the end of the stream.

The **getw** function has no boundary alignment requirements.

## SEE ALSO

C Library: *fdopen(), fopen(), getc(), putw()*

# gmtime

Break down a system-time value into units in the Greenwich Mean Time zone.

## SYNOPSIS

```
#include <time.h>
struct tm *gmtime(pclock)
     long     *pclock;
```

### Arguments

| | |
|---|---|
| <pclock> | The address of a system-time value |

### Returns

The address of the structure describing the system-time value

## DESCRIPTION

The **gmtime** function takes the system-time value referenced by the argument *<pclock>* and breaks it down into the year, month of the year (0-11), day of the month (1-31), day of the week (0-6, Sunday is 0), day of the year (0-365), hour (0-23), minute (0-59), and second (0-59). **Gmtime** saves that information in a structure and returns as its result the address of that structure.

The include-file *<time.h>* defines the structure referenced by the result of this function. That structure is:

```
        struct tm
        {
             int        tm_sec;
             int        tm_min;
             int        tm_hour;
             int        tm_mday;
             int        tm_mon;
             int        tm_year;
             int        tm_wday;
             int        tm_yday;
             int        tm_isdst;
        };
```

The *tm_sec* entry is the number of seconds into the minute and ranges from 0 to 59. *tm_min* is the number of minutes into the hour and ranges from 0 to 59. *tm_hour* is the number of hours into the day and ranges from 0 to 23. *tm_mday* is the day of the month and ranges from 1 to 31. *tm_mon* is the month of the year and ranges from 0 to 11. The *tm_year* entry is the number of years since 1900, *tm_wday* is the number of days into the week and ranges from 0 to 6, *tm_yday* is the number of days into the year and ranges from 0 to 365, and *tm_isdst* is always zero.

## NOTES

The system-time value is expressed in seconds since the epoch. The operating system defines the epoch as 00:00 (midnight) GMT, January 1, 1980.

The structure referenced by the result of this function is in static memory and is modified by subsequent calls to **ctime()**, **gmtime()**, or **localtime()**.

## SEE ALSO

C Library: *asctime(), ctime(), localtime()*

System Call: *time()*

Command: **date**

## gtty

Get the characteristics of an open character-device.

## SYNOPSIS

```
#include <errno.h>
#include <sys/sgtty.h>
int gtty(fildes, buf)
     int           fildes;
     struct sgttyb *buf;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor of the open character-device to examine |
| <buf> | The address of the structure to contain the characteristics of the character-device |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **gtty** function obtains the current characteristics of the open character-device referenced by the file descriptor *<fildes>* and writes information describing those characteristics into the structure referenced by *<buf>*. The **gtty** function returns zero as its result if it successfully obtains the characteristics of the open character-device. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **gtty** function fails if the file descriptor is out of range, does not reference an open file, or does not reference an open character-device. The include-file *<sys/sgtty.h>* contains the structure and data definitions for **gtty**.

The **gtty** function call expects *<buf>* to be the address of a structure that is defined as:

```
struct sgttyb
{
     unsigned char  sg_flag;
     unsigned char  sg_delay;
     unsigned char  sg_kill;
     unsigned char  sg_erase;
     unsigned char  sg_speed;
     unsigned char  sg_prot;
};
```

The bit-string *sg_flag* describes the current mode of the terminal. The values in the bit-string are:

| | |
|---|---|
| RAW | 0x01 |
| ECHO | 0x02 |
| XTABS | 0x04 |
| LCASE | 0x08 |
| CRMOD | 0x10 |
| SCOPE | 0x20 |
| CBREAK | 0x40 |
| CNTRL | 0x80 |

If RAW is set, the operating system considers the character-device to be in raw mode. In raw mode, the operating system suspends all processing of input and output. If clear, the operating system considers the character-device to be in non-raw mode (sometimes called *cooked mode*). In this mode, the operating system processes characters dependent upon the setting of the other bits in the bit-string.

If ECHO is set, the operating system echoes characters read to the character-device. If clear, the operating system does not echo characters to the device. If XTABS is set, the operating system expands tab-characters to spaces during output operations so that the next character written to the device is written to a column number that is an even multiple of eight. If XTABS is clear, the operating system writes tab-characters to the character-device with no expansion. Tab-characters are defined by the system to be 0x09 and are defined by the C compiler as '\t'.

If LCASE is set, the operating system changes all upper-case characters to lower-case characters during input operations and changes all lower-case characters to upper-case characters during output operations. If LCASE is clear, the operating system disables this feature. If LCASE mode is ignored, the LCASE function is replaced by the CAPS key on the keyboard. If CRMOD is set, the operating system writes a line-feed character to the character-device after every carriage-return character written. If CRMOD is clear, the operating system disables this feature.

If SCOPE is set, the operating system writes a backspace-character (0x08) followed by a space-character followed by another backspace-character to the character-device whenever a character-cancel character is read from the character-device. If SCOPE is clear, the operating system disables this feature. If CBREAK is set, the operating system considers the terminal to be in single-character mode. In this mode, the operating system reads data from the device one character at a time, passing each character to the calling task. If CBREAK is clear, the operating system considers the terminal to be in line mode, where it reads data from the device one line at a time, passing data to the calling task whenever a terminator is read.

If CNTRL is set, the operating system ignores all characters read from the character-device that are outside of the range 0x20 through 0x7E inclusive, except for the line-terminator character (carriage return), the keyboard-interrupt character (control-´c´), the quit-interrupt character (control-´\´), the character-cancel character, the line-cancel character, and the output-stop and output-start characters if any.

The bit-mask *sg_delay* indicates which characters, if written to the character-device, cause the operating system to pause before writing another character to the character-device. The values in that bit string are:

| | |
|---|---|
| DELNL | 0x03 |
| DELCR | 0x0C |
| DELTB | 0x10 |
| DELVT | 0x20 |
| DELFF | 0x20 |

These modes are ignored on the 4400 Series.

The *sg_kill* value defines line-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default line-cancel character is CTRL-u (0x15).

The *sg_erase* value defines the character-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default character-cancel character is the backspace-character (control-h, 0x08).

The bit-mask *sg_speed* contains configuration information for the character-device. Not all hardware supports the dynamic changing of the configuration. The values for the various configurations are:

| | | |
|---|---|---|
| D7S2EVEN | 0x00 | 7 data bits, 2 stop bits, even parity |
| D7S2ODD | 0x04 | 7 data bits, 2 stop bits, odd parity |
| D7S1EVEN | 0x08 | 7 data bits, 1 stop bit, even parity |
| D7S1ODD | 0x0C | 7 data bits, 1 stop bit, odd parity |
| D8S2NONE | 0x10 | 8 data bits, 2 stop bits, no parity |
| D8S1NONE | 0x14 | 8 data bits, 1 stop bit, no parity |
| D8S1EVEN | 0x18 | 8 data bits, 1 stop bit, even parity |
| D8S1ODD | 0x1C | 8 data bits, 1 stop bit, odd parity |
| CONFIG | 0x1C | mask for extracting configuration information |

The field *sg_prot* defines the type of start-stop protocol expected by the operating system for the character-device, and contains the baud rate used by the character-device. The values defined in that bit-string defining the protocol are:

| | |
|---|---|
| ESC | 0x80 |
| OXON | 0x40 |
| ANY | 0x20 |
| TRANS | 0x10 |
| IXON | 0X08 |
| BAUD_RATE | 0x0F |

If ESC is set, the operating system stops writing to the character-device when it reads an escape-character (0x1B) from the device. The operating system resumes writing to the character-device when it reads another escape-character from the device. If OXON is set, the operating system stops writing to the character-device when it reads an xoff-character (0x13). The operating system resumes writing to the character-device when it reads an xon-character (0x11). If ANY is set, the operating system uses any character read from the character-device as a substitute for the xon-character.

If TRANS is set, the operating system xon-xoff is transparent for raw mode (see the earlier discussion of the bit-string *sg-flag*). The 4400 Series ignores the IXON mode.

The baud rates are defined in the field as:

| | | |
|---|---|---|
| BAUD_RATE | 0x0F | baud-rate mask |
| B75 | 0x01 | 75 baud |
| B110 | 0x02 | 110 baud |
| B134 | 0x03 | 134.5 baud |
| B150 | 0x04 | 150 baud |
| B200 | 0x05 | 200 baud |
| B300 | 0x06 | 300 baud |
| B600 | 0x07 | 600 baud |
| B1200 | 0x08 | 1200 baud |
| B1800 | 0x09 | 1800 baud |
| B2400 | 0x0A | 2400 baud |
| B3600 | 0x0B | 3600 baud |
| B4800 | 0x0C | 4800 baud |
| B7200 | 0x0D | 7200 baud |
| B9600 | 0x0E | 9600 baud |
| B19200 | 0x0F | 19200 baud |

Not all hardware supports all of these baud rates and not all hardware allows the dynamic changing of baud rates.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode. |
| EINVAL | An argument to the function is invalid. |

| | |
|---|---|
| ENOTTY | The file is not a character device. |

## SEE ALSO

System Call: *creat(), dup(), dup2(), open(), pipe(), stty()*

Command: **commset, conset**

# idfd

Return the last file descriptor that signaled INPUT READY

## SYNOPSIS section

```
#include <errno.h>
int idfd()
```

### Arguments

None

### Returns

The file descriptor of the last file that sent the INPUT READY signal to the task or **-1** if no file has sent the signal.

## DESCRIPTION

The **idfd** function is used to interrogate the system to find out which file (file descriptor) caused the INPUT READY signal to be sent.

The INPUT READY signal can only be sent to a task for a file that has had the NOBLOCK mode set using the **fcntl()** function. The signal is only sent after a read from the file (normally a device) is unsuccessful because no data is available.

## ERRORS REPORTED

None

## NOTES

None

## SEE ALSO

C Library: *fcntl()*

## _ierrmsg

Initialize <*sys_errlist*> and <*sys_nerr*>.

## SYNOPSIS

```
void _ierrmsg()
```

### Arguments

None

### Returns

Void

## DESCRIPTIONS

The **ierrmsg** function initializes the global variable <*sys_nerr*> and the global table <*sys_errlist*> if they have not already been initialized.

## SEE ALSO

C Library: *errno, perror(), sys_errlist, sys_nerr*

# index

Find the first occurrence of a character in a character-string.

## SYNOPSIS

```
char *index(s, c)
     char     *s;
     char      c;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to search |
| <c> | The search character |

### RETURNS

The address of the first occurrence of the character in the string, or *(char *) NULL* if the string does not contain the character

## DESCRIPTION

The **index** function searches the character-string referenced by *<s>* for the first occurrence of the character *<c>*. If the string contains the character, the function returns as its result the address of the first occurrence of the character in the character-string. Otherwise, it returns *(char *) NULL.*

## NOTES

The **index** function is obsolete. It is only included for compatibility with older C libraries. New applications should use **strchr()**.

## SEE ALSO

C Library: *rindex(), strchr(), strrchr()*

# isalnum

Determine if a value is an alphabetic character or a decimal digit.

## SYNOPSIS

```
#include <ctype.h>
int isalnum(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is an alphabetic character or a decimal digit, zero otherwise

## DESCRIPTION

The **isalnum** function examines the value *<c>* and determines if it an alphabetic character or a decimal digit. Alphabetic characters are the characters ('A'-'Z') and ('a'-'z') inclusive. Decimal digits are the characters ('0'-'9') inclusive. If *<c>* is an alphabetic character or a decimal digit, the function returns a non-zero value, otherwise it returns zero.

## NOTES

The **isalnum** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isalpha

Determine if a value is an alphabetic character.

## SYNOPSIS

```
#include <ctype.h>
int isalpha(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

## RETURNS

Non-zero if the value is an alphabetic character, zero otherwise

## DESCRIPTION

The **isalpha** function examines the value *<c>* and determines if it an alphabetic character. Alphabetic characters are the characters (**A** through **Z**) and (**a** through **z**) inclusive. If *<c>* is an alphabetic character, **isalpha** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isalpha** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The *<c>* argument is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isascii

Determine if a value is an ASCII character.

## SYNOPSIS

```
#include <ctype.h>
int isascii(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

The **isascii** function returns **1** if *<c>* is a valid ASCII character, otherwise **isascii** returns **0**

## DESCRIPTION

The **isascii** function examines the value *<c>* and determines if it a valid ASCII character. Valid ASCII characters are the values between 0x00 and 0x7F (decimal values 0 through 127) inclusive. If *<c>* is a valid ASCII character, **isascii** returns **1**, otherwise it returns **0**.

## NOTES

The **isascii** function is implemented as a macro. However, it has no side-effects and produces a valid result for all values in the range of an *int*.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isatty

Determine if a file descriptor references a character-special file.

## SYNOPSIS

```
int isatty(fildes)
     int   fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor of an open file |

### Returns

1 if the open file is a character-special file, otherwise 0

## DESCRIPTION

This function examines the characteristics of the file referenced by the file descriptor *<fildes>*. If that file is a character-special file, this function returns 1 as its result. Otherwise it returns 0.

## NOTES

The function **fileno()** returns the file descriptor of an open stream.

A file descriptor is an index into the operating system's open file table. The system functions **creat()**, **dup()**, **dup2()**, **open()**, and **pipe()** return a file descriptor as their result.

## SEE ALSO

C Library: *fileno(), ttyname()*

System Call: *creat(), dup(), dup2(), open(), pipe(), ttyslot()*

# iscntrl

Determine if a value is a control character.

## SYNOPSIS

```
#include <ctype.h>
int iscntrl(c)
    int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a control character, zero otherwise

## DESCRIPTION

The **iscntrl** function examines *<c>* and determines if it a control character. Control characters are the values 0x00 through 0x1F inclusive and 0x7F. If *<c>* is a control character, **iscntrl** returns a non-zero value, otherwise it returns zero.

## NOTES

The **iscntrl** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isdigit

Determine if a value is a decimal digit.

## SYNOPSIS

```
#include <ctype.h>
int isdigit(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a decimal digit, zero otherwise

## DESCRIPTION

The **isdigit** function examines *<c>* and determines if it a decimal digit. Decimal digits are the characters ('0'-'9') inclusive. If *<c>* is a decimal digit, **isdigit** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isdigit** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

## isgraph

Determine if a value is a graphics character.

## SYNOPSIS

```
#include <ctype.h>
int isgraph(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a graphics character, zero otherwise

## DESCRIPTION

The **isgraph** function examines *<c>* and determines if it a graphics character. Graphic characters are alphabetic characters, decimal digits, and punctuation characters which are not white-space characters. If *<c>* is a graphics character, **isgraph** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isgraph** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# islower

Determine if a value is a lower-case alphabetic character.

## SYNOPSIS

```
#include <ctype.h>
int islower(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a lower-case alphabetic character, zero otherwise

## DESCRIPTION

The **islower** function examines *<c>* and determines if it a lower-case alphabetic character. Lower-case alphabetic characters are the characters ('a'-'z') inclusive. If *<c>* is a lower-case alphabetic character, **islower** returns a non-zero value, otherwise it returns zero.

## NOTES

The **islower** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isnan

Determine if a double precision floating point number is not-a-number.

## SYNOPSIS

```
include <math.h>
     int     isnan(x)     double     x;
```

### Arguments

None

### Returns

The value to examine Non-zero if the value is **not-a-number**, zero otherwise.

## DESCRIPTION

This function examines the value <*x*> and determines if it is **not-a-number. Not-a-number** is defined as a floating point number with an exponent of 2047 (maximum value) and any nonzero fraction.

## SEE ALSO

C Library: *finite(), matherr()*

# isprint

Determine if a value is a printable character.

## SYNOPSIS

```
#include <ctype.h>
int isprint(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a printable character, zero otherwise

## DESCRIPTION

The **isprint** function examines the value *<c>* and determines if it a printable character. Printable characters are alphabetic characters, decimal digits, and punctuation characters. If *<c>* is a printable character, **isprint** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isprint** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# ispunct

Determine if a value is a punctuation character.

## SYNOPSIS

```
#include <ctype.h>
int ispunct(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a punctuation character, zero otherwise

## DESCRIPTION

The **ispunct** function examines the value *<c>* and determines if it a punctuation character. Punctuation characters are all characters that are not alphabetic characters, decimal digits, white-space characters, or control characters. If *<c>* is a punctuation character, the function returns a non-zero value, otherwise it returns zero.

## NOTES

The **ispunct** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

# isspace

Determine if a value is a white-space character.

## SYNOPSIS

```
#include <ctype.h>
int isspace(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a white-space character, zero otherwise

## DESCRIPTION

The **isspace** function examines *<c>* and determines if it a white-space character. White-space characters are the space-character the horizontal-tab character ('\t'), the end-of-line character (EOL, '\r', '\n'), and the line-feed character (0x0A). If *<c>* is a white-space character, **isspace** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isspace** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

The C compiler translates the character '\n' to the line-feed character if the **cc** command is called with the +U option.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

Command: **cc**

# isupper

Determine if a value is an upper-case alphabetic character.

## SYNOPSIS

```
#include <ctype.h>
int isupper(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is an upper-case alphabetic character, zero otherwise

## DESCRIPTION

The **isupper** function examines the value *<c>* and determines if it an upper-case alphabetic character. Upper-case alphabetic characters are the characters (**A** through **Z**) inclusive. If *<c>* is an upper-case alphabetic character, **isupper** returns a non-zero value, otherwise it returns zero.

## NOTES

The **isupper** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if *<c>* is not a valid ASCII character or EOF.

The argument *<c>* is cast into an *int* if it is not already of that type.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

## isxdigit

Determine if a value is a hexadecimal digit.

### SYNOPSIS

```
#include <ctype.h>
int isxdigit(c)
     int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to examine |

### RETURNS

Non-zero if the value is a hexadecimal digit, zero otherwise

### DESCRIPTION

The **isxdigit** function examines the value <*c*> and determines if it a hexadecimal digit. Hexadecimal digits are the characters (**0** through **9**), (**a** through **f**), and (**A** through **F**) inclusive. If <*c*> is a hexadecimal digit, **isxdigit** returns a non-zero value, otherwise it returns zero.

### NOTES

The **isxdigit** function is implemented as a macro. It has no side-effects but its behavior is unpredictable if <*c*> is not a valid ASCII character or EOF.

The argument <*c*> is cast into an *int* if it is not already of that type.

### SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), toascii(), tolower(), _tolower(), toupper(), _toupper()*

## _itostr

Convert an *int* to a character-string.

## SYNOPSIS

```
char *_itostr(i, base, digits, psign)
     int       i;
     int       base;
     char     *digits;
     int      *psign;
```

### Arguments

| | |
|---|---|
| <i> | The value to convert |
| <base> | The base to use while converting |
| <digits> | The digits to use while converting |
| <psign> | The address of a flag to set to indicate the sign of the value or *(int *) NULL* if none |

### RETURNS

The address of the generated character-string

## DESCRIPTION

The **_itostr** function converts the *int* value *<i>* to its value represented in the base *<base>* using the digits in the character-string whose address is *<digits>*. If *<psign>* is *(int *) NULL*, the conversion is an unsigned conversion. Otherwise, the value referenced by *<psign>* is set to zero if *i* is equal to or greater than zero, non-zero otherwise.

The **_itostr** function returns as its result the address of the character-string it generated, or *(char *) NULL* if **_itostr** detects an error. Possible errors are a *<base>* less than or equal to one, or not enough digits in the character-string referenced by *digits* for the base.

## NOTES

The character-string referenced by the result is in static memory and is overwritten by subsequent calls to this or other conversion functions.

The longest character-string this function can generate is 32 characters.

## SEE ALSO

C Library: *atoi(), _ltostr()*

## kill

Send a signal to a task.

## SYNOPSIS

```
#include <errno.h>
#include <sys/signal.h>
int kill(taskid, signum)
     int        taskid;
     int        signum;
```

### Arguments

| | |
|---|---|
| <taskid> | The task-ID number of the task to receive the signal |
| <signum> | The signal to send the task |

### RETURNS

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **kill** function sends the signal numbered *<signum>* to the task whose task-ID number is *<taskid>*. A task may send a signal to another task only if its effective user is the system manager or it matches that of the specified task. The **kill** function returns zero if it successfully sent the task the specified signal, otherwise, it returns **-1** with *<errno>* set to the system error code.

The **kill** function fails if the signal number *<signum>* is out of range, there is no task with a task-ID number *<taskid>*, or the effective user of this task is not the system manager or does not match that of the specified task.

The include-file *<sys/signal.h>* defines these constants and their meaning:

| | | |
|---|---|---|
| SIGHUP | 1 | Hang-up |
| SIGINT | 2 | Keyboard |
| SIGQUIT | 3 | Quit |
| SIGEMT | 4 | EMT 0xA??? trap |
| SIGKILL | 5 | Task kill |
| SIGPIPE | 6 | Broken pipe |
| SIGTRACE | 8 | Trace |
| SIGTIME | 9 | Time limit |
| SIGALRM | 10 | Alarm |
| SIGTERM | 11 | Task terminate |
| SIGTRAPV | 12 | TRAPV instruction |
| SIGCHK | 13 | CHK instruction |
| SIGEMT2 | 14 | EMT 0xF??? emulation |
| SIGTRAP1 | 15 | TRAP #1 instruction |
| SIGTRAP2 | 16 | TRAP #2 instruction |
| SIGTRAP3 | 17 | TRAP #3 instruction |
| SIGTRAP4 | 18 | TRAP #4 instruction |
| SIGTRAP5 | 19 | TRAP #5 instruction |
| SIGTRAP6 | 20 | TRAP #6 instruction |
| SIGPAR | 21 | Parity error |
| SIGILL | 22 | Illegal instruction |
| SIGDIV | 23 | Division by zero |
| SIGPRIV | 24 | Privileged instruction |
| SIGADDR | 25 | Addressing  error |
| SIGDEAD | 26 | A child task has died |
| SIGWRIT | 27 | Write to read-only memory |
| SIGExEC | 28 | Execute from stack or data space |
| SIGBND | 29 | Segmentation violation |
| SIGUSR1 | 30 | User defined signal #1 |
| SIGUSR2 | 31 | User defined signal #2 |
| SIGUSR3 | 32 | User defined signal #3 |
| SIGABORT | 33 | Program abort |
| SIGSPLR | 34 | Spooler signal |
| SIGINPUT | 35 | Input is ready |
| SIGDUMP | 36 | Take memory dump |
| SIGVEN14 | 62 | Millisecond alarm |
| SIGVEN15 | 63 | Mouse/keyboard event interrupt |

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager or the it does not match that of the specified task |
| EINVAL | The signal number is out of range |
| ESRCH | Invalid task number |

## SEE ALSO

System Call: *signal()*

Command: **int**

# _l2tos

Convert two-byte integers to *short* integers.

## SYNOPSIS

```
void _l2tos(sp, cp, n)
     short    *sp;
     char     *cp;
     int       n;
```

### Arguments

| | |
|---|---|
| <sp> | The address of the buffer to contain the *short* integers |
| <cp> | The address of the buffer containing the two-byte integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The **_l2tos** function converts *<n>* two-byte integers packed in the array of *char* referenced by *<cp>*, saving the converted values in the array of *short* referenced by *<sp>*. The **_l2tos** function returns no result.

## NOTES

The **_l2tos** function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *l3tol(), _l4tol(), ltol3(), _ltol4(), _stol2()*

## l3tol

Convert three-byte integers to *long* integers.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <lp> | The address of the buffer to contain the *long* integers |
| <cp> | The address of the buffer containing the three-byte integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The **l3tol** function converts *<n>* three-byte integers packed in the array of *char* referenced by *<cp>*, saving the converted values in the array of *long* referenced by *<lp>*. The **l3tol** function returns no result.

## NOTES

The **l3tol** function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *_l2tos(), _l4tol(), ltol3(), _ltol4(), _stol2()*

# _l4tol

Convert four-byte integers to *long* integers.

## SYNOPSIS

```
void _l4tol(lp, cp, n)
     long     *lp;
     char     *cp;
     int       n;
```

### Arguments

| | |
|---|---|
| <lp> | The address of the buffer to contain the *long* integers |
| <cp> | The address of the buffer containing the four-byte integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The **l4tol** function converts *<n>* four-byte integers packed in the array of *char* referenced by *<cp>*, saving the converted values in the array of *long* referenced by *<lp>*. The **l4tol** function returns no result.

## NOTES

The **l4tol** function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *_l2tos(), l3tol(), ltol3(), _ltol4(), _stol2()*

# ldexp

Generate a floating-point value from a mantissa and an exponent.

## SYNOPSIS

```
double ldexp(fp, exp)
     double    fp;
     int       exp;
```

### Arguments

| | |
|---|---|
| <fp> | The mantissa |
| <exp> | The exponent |

### Returns

The floating-point value represented by the exponent and the mantissa

## DESCRIPTION

The **ldexp** function generates a floating-point value by applying the exponent *<exp>* to the mantissa *<fp>*. The result is calculated by multiplying the mantissa *<fp>* by the result of raising 2 to the power indicated by *<exp>*. **Ldexp** returns the generated value as its result.

All floating-point values are represented by a mantissa and an exponent. A non-zero value is represented by a mantissa with an absolute value greater than or equal to 0.5 and less than 1.0 and an exponent that is a signed integer. The floating-point value represented by the mantissa and exponent is 2 raised to the power indicated by the exponent which is then multiplied by the mantissa. For example, the floating-point value 1.0 is represented by a mantissa of 0.5 and an exponent of 1. A floating-point 0.0 is represented by a mantissa of 0.0 and an exponent of 0.

## SEE ALSO

C Library: *frexp(), modf()*

# link

Create a link to a file.

## SYNOPSIS

```
#include <errno.h>
int link(path, newlink)
     char     *path;
     char     *newlink;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname for an existing file |
| <newlink> | The address of a character-string containing the pathname of the link to create |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **link** function establishes a link to the file reached by the pathname in the character-string referenced by *<path>* called by the pathname in the character-string referenced by *<newlink>*. The pathname *<path>* must exist and the file it reaches cannot be a directory. The *<newlink>* pathname must not exist. The device for *<newlink>* must be the same as that for *<path>*. The directory for *<newlink>* must give the current effective user writing permission. The maximum number of links allowed is 127.

The **link** function returns zero as its result if it successfully establishes the link. Otherwise, **link** returns **-1** with *<errno>* set to the system error code. The **link** function fails if:

- the path in *<path>* or *<newlink>* can not be followed or contains a file that is not a directory
- the pathname *<path>* does not exist
- the pathname *<newlink>* already exists
- the file reached by *<path>* is a directory
- the directory for *<newlink>* does not grant the current effective user writing permission
- the link crosses devices

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The directory for <*newlink*> does not give the current effective user writing permission |
| EEXIST | The pathname <*newlink*> already exists |
| EISDR | The file reached by <*path*> is a directory and the current effective user is not the system manager |
| EMSDR | The path can not be followed for either <*path*> or <*newlink*> |
| ENOENT | The pathname <*path*> does not exist |
| ENOTDIR | The path in either <*path*> or <*newlink*> contains a file that is not a directory |
| EXDEV | Attempting to link across devices |

## NOTES

Linking to a file changes the last-access time of that file.

## SEE ALSO

System Call: *fstat(), stat(), unlink()*

Command: **link, remove, rename**

## localtime

Break down a system-time value into units in the local time zone.

## SYNOPSIS

```
#include <time.h>
struct tm *localtime(pclock)
     long     *pclock;
```

### Arguments

| | |
|---|---|
| <pclock> | The address of a system-time value |

### Returns

The address of the structure describing the system-time value

## DESCRIPTION

The **localtime** function takes the system-time value referenced by the argument *<pclock>* and breaks it down into the year, month of the year (0-11), day of the month (1-31), day of the week (0-6, Sunday is 0), day of the year (0-365), hour (0-23), minute (0-59), and second (0-59) in the current time zone, applying the standard U. S. A. daylight-savings time conversion if necessary. The **localtime** function saves that information in a structure and returns as its result the address of that structure.

The include-file *<time.h>* defines the structure referred to by the result of this function. That definition is:

```
struct tm
{
     int       tm_sec;
     int       tm_min;
     int       tm_hour;
     int       tm_mday;
     int       tm_mon;
     int       tm_year;
     int       tm_wday;
     int       tm_yday;
     int       tm_isdst;
};
```

The *tm_sec* entry is the number of seconds into the minute and ranges from 0 to 59. *tm_min* is the number of minutes into the hour and ranges from 0 to 59. *tm_hour* is the number of hours into the day and ranges from 0 to 23. *tm_mday* is the day of the month and ranges from 1 to 31. *tm_mon* is the month of the year and ranges from 0 to 11. The *tm_year* entry is the number of years since 1900. *tm_wday* is the number of days into the week and ranges from 0 to 6. *tm_yday* is the number of days into the year and ranges from 0 to 365. *tm_isdst* is one if the standard U. S. A. daylight-savings time conversion was applied, zero otherwise.

## NOTES

The system time value is expressed in seconds since the epoch. The operating system defines the epoch as 00:00 (midnight) GMT, January 1, 1980.

The structure referenced by the result of this function is in static memory and is modified by subsequent calls to **ctime()**, **gmtime()**, or **localtime()**.

The **localtime** function applies the standard U. S. A. daylight-savings time conversion only if the current system configuration indicates that daylight-savings time is in effect. If standard U. S. A. daylight-savings time is in effect, **localtime** adds an hour to the time if the time falls between 02:00 AM on the last Sunday in April and 01:00 AM on the last Sunday in October.

The **localtime** function calls **tzset()** if necessary, setting the global variables *daylight*, *timezone*, and *tzname*.

## SEE ALSO

C Library: *asctime(), ctime(), daylight, gmtime(), timezone, tzname, tzset()*

System Call: *time()*

Command: **date**

# lock

Lock a task in memory or unlock a locked task.

## SYNOPSIS

```
#include <errno.h>
int lock(flag)
    int       flag;
```

### Arguments

| | |
|---|---|
| <flag> | A flag that indicates lock or unlock |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

If the value *<flag>* is not zero, the **lock** function locks the current task in memory, preventing the operating system from swapping the task to the system swap space. Otherwise, **lock** unlocks the current task, permitting the operating system to swap the task to the system swap space if necessary. The current effective user must be the system manager.

The **lock** function returns zero if it succeeds, otherwise, it returns **-1** with *<errno>* set to the system error code. The **lock** function fails if the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |

## NOTES

Unlocking a task that is not locked is not an error.

## SEE ALSO

System Call: *memman()*

# log

Calculate the natural logarithm of a value.

## SYNOPSIS

```
#include <math.h>
double log(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value with the natural logarithm to be computed |

### Returns

The natural logarithm of the argument *<x>*

## DESCRIPTION

The **log** function calculates the natural logarithm of *<x>*. The natural logarithm of *<x>* is defined as the value that e (2.718281828459 . . . ) must be raised to generate the value *<x>*. The **log** function returns the calculated value as its result.

The **log** function demands that *<x>* be a value greater than 0.0. Values less than or equal to 0.0 cause a domain error. If the function detects a domain error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. It sets the *<type>* element of the structure to the constant DOMAIN, *<name>* to the address of the character-string *<log>*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the function writes the message

```
        log() error:  Non-positive argument
```

to the standard error I/O stream *<stderr>* and sets *<errno>* to EDOM. The return value, which is system-dependent, is given in the manual page for the **kill** command. If **matherr()** returns something other than zero, it returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *exp(), log10(), matherr()*

## log10

Calculate the base-10 logarithm of a value.

## SYNOPSIS

```
#include <math.h>
double log10(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value whose base-10 logarithm is to be computed |

### Returns

The base-10 logarithm of the argument *<x>*

## DESCRIPTION

The **log10** function calculates the base-10 logarithm of *<x>*. The base-10 logarithm of *<x>* is defined as the value that 10.0 must be raised to generate *<x>*. The **log10** function returns the calculated value as its result.

The **log10** function demands that *<x>* be greater than 0.0. Values less than or equal to 0.0 cause a domain error. If **log10** detects a domain error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. **Matherr** sets the *<type>* element of the structure to DOMAIN, *<name>* to the address of the character-string *log10*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, the **log10** function writes the message

```
        log10() error:  Non-positive argument
```

to the standard error I/O stream *<stderr>* and sets *<errno>* to EDOM. The return value, which is system-dependent, is given in the manual page for the **kill** command. If **matherr()** returns something other it returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *exp(), log(), matherr()*

# longjmp

Perform a non-local goto.

## SYNOPSIS

```
#include <setjmp.h>
void longjmp(env, val)
     jmp_buf   env;
     int       val;
```

### Arguments

| | |
|---|---|
| <env> | Contains environmental information about the target of the non-local goto |
| <val> | The value to return as the apparent result of the **setjmp()** associated with *<env>* |

### Returns

Never

## DESCRIPTION

The **longjmp** function restores the program execution environment to that described by the argument *<env>*. The effect is that of a *goto* to the **setjmp()** call, which saved the environmental information in the argument *<env>*, if *<val>* is not zero or 1 otherwise as the apparent result of the **setjmp()** call.

## NOTES

Statements following the call to **longjmp** are never executed.

The scope containing the **setjmp()** call that set up the *<env>* argument must not have executed a return or the result of this function is unpredictable.

All variables allocated to a register are restored to their value at the **setjmp()** call.

## SEE ALSO

C Library: *setjmp()*

# lrec

Add an entry to the lock table of the operating system.

## SYNOPSIS

```
#include <errno.h>
int lrec(fildes, count)
     int       fildes;
     int       count;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor for the file containing the record to lock |
| <count> | The number of bytes to lock from the current file position |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **lrec** function adds an entry to the operating system's lock table for the open file referenced by the file descriptor *<fildes>*, locking a record beginning at the current file position containing *<count>* bytes. If the current task has an existing entry in the lock table for the same file descriptor, **lrec** removes that entry. The **lrec** function returns zero as its result if it successfully locks the record, otherwise it returns **-1** with *<errno>* set to the system error code.

The **lrec** function fails if the lock table of the operating system contains an entry made by another task for the file referenced by *fildes* and the record locked by that entry contains all or part of the record this function is trying to lock, or lock table is full. The **lrec** function also fails if the file descriptor *<fildes>* is out of range, does not reference an open file, or references a file that is not a regular file.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file, or references a pipe, character-special file (character-device), or a block-special file (block-device). |
| EINVAL | An argument to the function is invalid. |
| ELOCK | The record specified could not be locked because there already exists a lock on all or part of that record, or the operating system's lock table is full. |

## NOTES

Locking a record only prevents others from locking it. This and other tasks may read or modify the record and may alter the file containing the record.

The **lrec** function removes any existing lock table entry made by the current task for the specified file without regard to the eventual outcome of the function.

The **lrec** function only permits one entry in the system lock table for each file a task has open.

The operating system removes all lock table entries made by a task when that task terminates.

## SEE ALSO

System Call: *creat(), dup(), dup2(), open(), pipe(), urec()*

# lseek

Change the current file position of an open file.

## SYNOPSIS

```
#include <errno.h>
long lseek(fildes, offset, type)
     int       fildes;
     long      offset;
     int       type;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor of the file to reposition |
| <offset> | A count describing the offset of the new position |
| <type> | A value describing the offset type |

### Returns

The new offset from the beginning of the file if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **lseek** function changes the current file position in the file descriptor *<fildes>*, dependent upon the *<offset>* and *<type>* values. If *<type>* is 0, **lseek** interprets *<offset>* as an absolute byte-count from the beginning of the file. If *<type>* is 1, **lseek** interprets *<offset>* as a byte-count relative to the current file position. If *<type>* is 2, **lseek** interprets *<offset>* as a byte-count relative to the end of the file. If **lseek** successfully changes the current file pointer, **lseek** returns the new file position, which is the offset relative to the beginning of the file. Otherwise, **lseek** returns **-1** with *<errno>* set to the system error code.

The **lseek** function fails if the file descriptor *<fildes>* is out of range, does not reference an open file, or references a file that can not be repositioned, such as a pipe or a character-special file. **Lseek** also fails if the requested position is before the beginning of the file or the value *<type>* is not valid.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference and open file or the file is not open in the proper mode. |
| EINVAL | The value <*type*> is not valid or the <*fildes*> descriptor is out of range . |
| ESEEK | The requested file position is before the beginning of the file or the *fildes* file descriptor references a file that can not be repositioned. |

## NOTES

The function call **lseek**(<*fildes*>,(*long*)**0,1**) returns as its result the current position of the file.

The **lseek** function does not change the current position of the file if the function reports an error.

If the new position is beyond the current end of the file, the **lseek** function creates a gap in the file that contains zeros if read. The **lseek** function does not allocate any new blocks of media if the file resides on a block-device.

## SEE ALSO

C Library: *fseek()*

System Call: *creat(), dup(), dup2(), open(), pipe()*

# ltol3

Convert *long* integers to three-byte integers.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <cp> | The address of the buffer to contain the three-byte integers |
| <lp> | The address of the buffer containing the *long* integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The **ltol3** function converts *<n>* *long* integers in the array referenced by *<lp>* to three-byte integers, saving the converted values packed into the array of *char* referenced by *<cp>*. The **ltol3** function returns no result.

## NOTES

The **ltol3** function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *_l2tos(), l3tol(), _l4tol(), _ltol4(), stol2()*

# _ltol4

Convert *long* integers to four-byte integers.

## SYNOPSIS

```
    d _ltol4(cp, lp, n)

    int       n;
```

### Arguments

| | |
|---|---|
| <cp> | The address of the buffer to contain the four-byte integers |
| <lp> | The address of the buffer containing the *long* integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The _ltol4 function converts *<n>* *long* integers in the array referenced by *<lp>* to four-byte integers, saving the converted values packed into the array of *char* referenced by *<cp>*. The function returns no result.

## NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *_stol2(), _l2tos(), l3tol(), _l4tol(), ltol3(), stol2()*

# _ltostr

Convert a *long* to a character-string.

## SYNOPSIS

```
char *_ltostr(i, base, digits, psign)
     long      i;
     int       base;
     char     *digits;
     int      *psign;
```

### Arguments

| | |
|---|---|
| <i> | The value to convert |
| <base> | The base to use while converting |
| <digits> | The digits to use while converting |
| <psign> | The address of a flag to set to indicate the sign or *(int *) NULL* |

### Returns

The address of the generated character-string

## DESCRIPTION

The **_ltostr** function converts the *long* value *<i>* to its value represented in the base *<base>* using the digits in the character-string referenced by *<digits>*. If *<psign>* is *(int *) NULL*, the conversion is an unsigned conversion. Otherwise, the value referenced by *<psign>* is set to zero if *i* is equal to or greater than zero, non-zero otherwise.

The **_ltostr** function returns as its result the address of the character-string it generated, or *(char *) NULL* if **_ltostr** detected an error. Possible errors are a *<base>* less than or equal to one, or not enough digits in the character-string referenced by *<digits>* for the base *<base>*.

## NOTES

The character-string referenced by the result is in static memory and is overwritten by subsequent calls to this or other conversion functions.

The longest character-string **_ltostr** can generate is 32 characters.

## SEE ALSO

C Library: *atol(), _itostr()*

## make_realtime

Declare the task to be a real-time task.

## SYNOPSIS

```
#include <errno.h>
int make_realtime(incr)
     int  incr;
```

### Arguments

| | |
|---|---|
| <incr> | The value to add to the task's priority |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

When called with a nonzero increment *<incr>*, the **make_realtime** function makes the task a real-time task. A real-time task has priority over all non-real-time tasks. The **make_realtime** function changes the scheduling priority of the task by adding the signed increment to the value representing its priority. The modified priority is used by the system scheduler when scheduling the CPU among several real-time tasks. When called with an increment of zero, this function makes the task a non-real-time task. The function returns zero if it successfully makes the task a real-time or non-real-time task, otherwise it returns **-1** with *<errno>* set to the system error code.

The **make_realtime** function fails if the current effective user is not the system manager.

This function may not be available on all versions of the operating system.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EBDCL | The real-time feature is not available |

## NOTES

The function does not change the scheduling priority if the increment *<incr>* causes the priority to exceed the maximum or to be less than the minimum.

The higher the value representing the priority of the task, the lower the priority of the task.

## SEE ALSO

Commands: **make_realtime**

# malloc

Allocate memory.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <nbytes> | The number of bytes to allocate |

### Returns

The address of the allocated block of memory or *(char *) NULL* if none was available

## DESCRIPTION

The **malloc** function allocates *<nbytes>* bytes of memory from the arena of available memory. **Malloc** returns the address of the first byte of the allocated memory or *(char *) NULL* if none was available.

The first byte of the allocated memory is properly aligned for any use.

## NOTES

The **malloc** function **free()** returns allocated memory to the arena of available memory.

## SEE ALSO

C Library: *calloc(), free(), realloc()*

System Call: *brk(), cdata(), sbrk()*

# matherr

Floating-point error-handling function for built-ins.

## SYNOPSIS

```
#include <math.h>
int matherr(ptr)
     struct exception  *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of a structure containing information about the function and the arguments to the function that detected the error |

### Returns

Zero if the function detecting the error is to perform its standard function, non-zero if the function is to proceed using as its the return value specified in the structure passed to this function

## DESCRIPTION

The **matherr** function provides the programmer with a method of intercepting errors and exceptional conditions detected by a floating-point builtin before the builtin reports the error. All builtins call **matherr** immediately before reporting an error. If the **matherr** function returns zero, the calling function handles the error normally. Otherwise, the calling function uses the return value found in the structure referenced by *<ptr>* as its result and proceeds as if the error had not occurred.

If no **matherr()** function is provided by the programmer, the builtin library provides this default function:

```
int matherr(x)
     struct exception  *x;
{
     return(0);
}
```

This causes all of the floating-point builtins to handle errors normally. By including their own version of this function, the user can intercept all floating-point errors detected by floating-point builtins, determine what function detected the error, examine the arguments that caused the error, and alter the behavior of the builtin, including the value returned by the function. For example, the following version changes the way **pow()** reports an error so that a singularity error returns zero but does not write a message. It instructs the builtins to handle all other errors normally.

```
int matherr(x)
     struct exception  *x;
{
     if ((strcmp(x->name, "pow") == 0) &&
         (x->type == SING))
     {
          x->retval = 0.0;
          return(1);
     }
     return(0);
}
```

The structure referenced by *<ptr>* is defined as:

```
struct exception
{
     int        type;
     char      *name;
     double     arg1;
     double     arg2;
     double     retval;
}
```

The element *<type>* describes the type of error, *<name>* is the address of a character-string containing the name of the function reporting the error, *<arg1>* is the first argument to that function, *<arg2>* is the second argument to that function, if any, and *retval* is the value to use as the result of that function if **matherr** returns something other than zero. The include-file *<math.h>* defines this structure.

The *<type>* element in the above structure describes the type of error. The include-file *<math.h>* also defines these constants. The *<type>* element is one of these values:

| | |
|---|---|
| DOMAIN | A DOMAIN error indicates that the functions arguments are out of the domain of the function. |
| OVERFLOW | An OVERFLOW error indicates that the result of the function is larger than can be represented by a *double.* |
| PLOSS | A PLOSS error indicates that the result of the function reflects a partial loss of significance. |
| SING | A SING error indicates that the particular argument or arguments presented to the function are not permitted by the function. |
| TLOSS | A TLOSS error indicates that the result of the function reflects a total loss of significance. |
| UNDERFLOW | An UNDERFLOW error indicates that the magnitude of the result of the function is smaller than can be represented by a *double.* |

## SEE ALSO

C Library: *acos() asin(), atan2(), exp(), log(), log10(), pow(), sqrt(), tan()*

# memccpy

Copy memory.

## SYNOPSIS

```
#include <memory.h>
char *memccpy(ptr1, ptr2, c, n)
     char     *ptr1;
     char     *ptr2;
     int       c;
     int       n;
```

### Arguments

| | |
|---|---|
| <ptr1> | The target buffer address |
| <ptr2> | The source buffer address |
| <c> | The stop-value |
| <n> | The maximum number of bytes to copy |

### Returns

The address of the byte following the copy of the stop-value *<c>* in the target buffer, or *(char *) NULL* if *<c>* was not found

## DESCRIPTION

The **memccpy** function copies bytes from the buffer with the address *<ptr2>* to the buffer with the address *<ptr1>* until either the value *<c>* is copied or the requested number of bytes has been copied, whichever comes first. **Memccpy** returns the address of the byte following the copy of the value *<c>* in the target buffer, or *(char *) NULL* if that value was not found.

## NOTES

The behavior of overlapping copy operations is not defined and may behave differently on different systems.

If *<n>* is less than or equal to zero, the function copies no data and returns *(char *) NULL* as its result.

The include-file *<memory.h>* defines this and other block memory functions.

## SEE ALSO

C Library: *memchr(), memcmp(), memcpy(), memset()*

# memchr

Find a value in a block of memory.

## SYNOPSIS

```
#include <memory.h>
char *memchr(ptr, c, n)
     char     *ptr;
     int       c;
     int       n;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the buffer to search |
| <c> | The value to search for |
| <n> | The maximum number of bytes to search |

### Returns

The address of the first byte with the value *<c>*, or *(char *) NULL* if *<c>* was not found

## DESCRIPTION

The **memchr** function searches the first *<n>* bytes in the buffer with the address *<ptr>* for the value *<c>*. If the value is found, **memchr** returns the address of that value as its result. Otherwise, it returns *(char *) NULL.*

## NOTES

If *n* is less than or equal to zero, **memchr** always returns *(char *) NULL.*

The include-file *<memory.h> defines this and other block memory functions.*

## SEE ALSO

C Library: *memccpy(), memcmp(), memcpy(), memset()*

# memcmp

Compare two blocks of memory.

## SYNOPSIS

```
#include <memory.h>
int memcmp(ptr1, ptr2, n)
     char     *ptr1;
     char     *ptr2;
     int       n;
```

### Arguments

| | |
|---|---|
| <ptr1> | The address of the first buffer to compare |
| <ptr2> | The address of the second buffer to compare |
| <n> | The maximum number of bytes to compare |

### Returns

A value less than, equal to, or greater than zero, if the buffer referenced by *<ptr1>* is lexicographically less than, equal to, or greater than the buffer referenced by *<ptr2>*.

## DESCRIPTION

The **memcmp** function lexicographically compares the buffer referenced by *<ptr1>* with the buffer referenced by *<ptr2>* and returns as its result a value which indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the buffer referenced by *<ptr1>* is lexicographically less than, equal to, or greater than the buffer referenced by *<ptr2>*.

## NOTES

If <*n*> is less than or equal to zero, the result of this function is always zero.

A non-zero result is the result of subtracting the differing character in the buffer referenced by <*ptr2*> from the differing character in the buffer referenced by <*ptr1*>.

The include-file <*memory.h*> defines this and other block memory functions.

## SEE ALSO

C Library: *memccpy(), memchr(), memcpy(), memset()*

# memcpy

Copy memory.

## SYNOPSIS

```
#include <memory.h>
char *memcpy(ptr1, ptr2, n)
     char     *ptr1;
     char     *ptr2;
     int       n;
```

### Arguments

| | |
|---|---|
| <ptr1> | The target buffer address |
| <ptr2> | The source buffer address |
| <n> | The number of bytes to copy |

### Returns

*<ptr1>*

## DESCRIPTION

The **memcpy** function copies bytes from the buffer with the address *<ptr2>* to the buffer with the address *<ptr1>* until the requested number of bytes has been copied. **Memcpy** returns as its result the address of the target buffer *<ptr1>*.

## NOTES

The behavior of overlapping copy operations is not defined and may behave differently on different systems.

If *<n>* is less than or equal to zero, the function copies no data.

The include-file *<memory.h>* defines this and other block memory functions.

## SEE ALSO

C Library: *memccpy(), memchr(), memcmp(), memset()*

## memman

Perform a memory management operation.

## SYNOPSIS

```
#include <errno.h>
int memman(fcn, loaddr, hiaddr)
     int       fcn;
     char      *loaddr;
     char      *hiaddr;
```

### Arguments

| | |
|---|---|
| <fcn> | A value indicating the memory management function to perform |
| <loaddr> | The lowest address of memory to affect by the function |
| <hiaddr> | The highest address of memory to affect by the function |

### Returns

The **memman** function returns zero if successful, otherwise **memman** returns **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **memman** function performs a memory management operation on the region of memory whose lowest address is *<loaddr>* and whose highest address is *<hiaddr>*. The value *<fcn>* selects which operation the function performs. The operations performed by this function are machine-dependent and may be different for the various implementations of the operating system. The **memman** function expects the current effective user to be the system manager. **Memman** returns zero if it successfully performs the memory management function on the specified region of memory. Otherwise, **memman** returns **-1** with *<errno>* set to the system error code.

The **memman** function fails if the function code *<fcn>* is out of range, if the high memory address *<hiaddr>* is lower than the low memory address *<loaddr>*, or the current effective user is not the system manager. The **memman** function may also fail for reasons peculiar to the machine-dependent implementation of the function.

The **memman** function has the following operations:

```
0    Clear the region's dirty-bit
1    Lock the region in memory
2    Unlock the region
3    Set write-protection on the region
4    Remove write-protection from the region
5    Release the memory allocated to the region
```

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EINVAL | The function type is invalid or the starting address <*loaddr*> is higher than the ending address <*hiaddr*> |
| EVFORK | The task shares its memory with its parent and may not call this function |

# memset

Set a block of memory.

## SYNOPSIS

```
#include <memory.h>
char *memset(ptr, c, n)
     char     *ptr;
     int       c;
     int       n;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of the buffer to set |
| <c> | The value to set |
| <n> | The number of bytes to set |

### Returns

*<ptr>*

## DESCRIPTION

The **memset** function sets *<n>* bytes of memory beginning at the address *<ptr>* to the value *<c>*. It returns as its result its argument *<ptr>*.

## NOTES

If *<n>* is less than or equal to zero, **memset** modifies no memory.

The include-file *<memory.h>* defines this and other block memory functions.

## SEE ALSO

C Library: *memccpy(), memchr(), memcmp(), memcpy()*

# mknod

Add an entry to the file-system that is a directory, a character-special file, or a block-special file.

## SYNOPSIS

```
#include <errno.h>
int mknod(path, desc, devnum)
     char     *path;
     short     desc;
     short     devnum;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing a pathname to the entry to create |
| <desc> | A bit-string describing the type of entry to create and the access permissions to assign to it |
| <devnum> | The major and minor device numbers to assign to the character-special or block-special file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **mknod** function adds an entry to the file-system that is a directory, a character-special file, or a block-special file. It gives the new entry the name found in the character-string referenced by *<path>*. The **mknod** function determines the type of entry it creates from the bit-string *<desc>*. **Mknod** assigns to that entry the access permissions described by the bit-string *<desc>*, and if the entry is a character-special or block-special file, **mknod** assigns to it the major and minor device numbers defined in the value *<devnum>*. The function ignores the argument *<devnum>* if it is creating a directory. **Mknod** requires that the current effective user be the system manager. **Mknod** returns zero if it successfully creates the entry in the file-system. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **mknod** function fails if the pathname already exists, the path can not be followed, the path contains a file which is not a directory, or the disk is full. The **mknod** function also fails if either the *desc* or *<devnum>* arguments are invalid, or the current effective user is not the system manager.

The bit-string *<desc>* describes the file type of the new entry and its access permissions. The *<desc>* bit-string is defined as:

| | |
|---|---|
| `0x0001` | `Grant reading permission to the file's owner` |
| `0x0002` | `Grant writing permission to the file's owner` |
| `0x0004` | `Grant execution (or searching) permission to the file's owner` |
| `0x0008` | `Grant reading permission to other users` |
| `0x0010` | `Grant writing permission to other users` |
| `0x0020` | `Grant execution (or searching) permission to other users` |
| `0x0040` | `Give the task executing the file the access permissions of the owner of the file` |
| `0x0200` | `Make the file a block-special file` |
| `0x0400` | `Make the file a character-special file` |
| `0x0800` | `Make the file a directory` |

The **mknod** function requires that the bit-string *<desc>* contain exactly one of the bit-values describing the type of file, a directory, a character-special file, or a block-special file. **Mknod** allows the bit-string to contain any of the bit-values defining the permissions, in any combination.

The *<devnum>* argument contains the major and minor device numbers to assign to the character-special or block-special file. The most-significant byte contains the major device number, the least-significant byte contains the minor device number. This argument is ignored if the function creates a directory.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EEXIST | The pathname already references a file |
| EINVAL | The file description *<desc>* or the device number *<devno>* is not valid |
| EMSDR | The function could not follow the path to the file |
| ENOSPC | The device is full |
| ENOTDIR | A part of the path is not a directory |

## NOTES

A character-special file is usually attached to a character-oriented device. Likewise, a block-special file is usually attached to a block-oriented device.

The third argument *devnum* should be specified as zero if <*desc*> indicates that the new entry is a directory.

## SEE ALSO

System Call: *creat()*

Command: **crdir, makdev**

# mktemp

Generate a unique pathname from a template.

## SYNOPSIS

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

### Arguments

<template> The address of the character-string containing the template for the temporary pathname

### Returns

The argument *<template>* if **mktemp** successfully generates a unique pathname, *(char *) NULL* otherwise

## DESCRIPTION

The **mktemp** function generates a unique pathname from the template pathname in the character-string referenced by *<template>*. A unique pathname is one that does not reach a file but contains a path that can be followed. **Mktemp** returns *<template>* if it successfully generated a pathname for a file that does not exist, or *(char *) NULL* otherwise.

If the template pathname ends in six *x* characters, it replaces those characters with an *A* followed by the five-character representation of the current process-ID. It then checks the filesystem for that pathname. If that pathname does not exist, **mktemp** returns *<template>*. If that pathname already exists, it changes the *A* to a *B* and retrys, continuing until it generates a pathname that does not reference an existing file or it exhausts the upper- and lower-case alphabet.

If the template pathname does not end in *x* or *X* characters, **mktemp** returns *<template>* if the pathname is unique, or *(char *) NULL* if it is not.

## NOTES

If **mktemp** cannot follow the path in the template pathname, or contains a file that is not a directory, the function returns *(char *) NULL.*

If **mktemp** returns *(char *) NULL*, the variable *<errno>* contains the system error code describing the error.

## SEE ALSO

System Call: *getpid(), read()*

## modf

Separate a floating-point value into its integral and fractional parts.

## SYNOPSIS

```
double modf(fp, dptr)
     double    fp;
     double   *dptr;
```

### Arguments

| | |
|---|---|
| <fp> | The floating-point value to separate |
| <dptr> | The address of the *double* to receive the integral part of the floating-point value *<fp>* exponent |

### Returns

The fractional part of the floating-point value *<fp>*

## DESCRIPTION

The **modf** function separates the fractional part of the floating-point value *<fp>* from its integral part. The **modf** function stores the integral part through *<dptr>* and returns the fractional part as its result.

If the absolute value of *<fp>* is less than 1.0, **modf** stores 0.0 through *<dptr>* and returns *<fp>*. If *<fp>* contains no fractional part, the function stores *<fp>* through *<dptr>* and returns 0.0.

## SEE ALSO

C Library: *frexp(), ldexp()*

# mount

Mount a block-special file onto the file-system.

## SYNOPSIS

```
#include <errno.h>
int mount(spcnam, dirnam, rwflag)
     char     *spcnam;
     char     *dirnam;
     int       rwflag;
```

### Arguments

| | |
|---|---|
| <spcnam> | The address of a character-string containing a pathname to the block-special file to mount |
| <dirnam> | The address of a character-string containing a pathname to the directory to mount the block-special file |
| <rwflag> | A value indicating the type of accessing to permit |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **mount** function mounts the block-special file reached by the pathname in the character-string referenced by *<spcnam>* onto the directory reached by the pathname in the character-string referenced by *<dirnam>*. If the value *<rwflag>* is 0, **mount** mounts the file, permitting reading and writing access. If *rwflag* is not 0, **mount** mounts the file so that it does not permit writing access. The **mount** function returns zero if it successfully mounts the specified block-special file onto the specified directory. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **mount** function fails if it can not follow the path in *<spcnam>* or *<dirnam>*, the path in *<spcnam>* or *<dirnam>* contains a file which is not a directory, or either *<spcnam>* or *<dirname>* do not exist. **Mount** also fails if:

- the file reached by *<spcnam>* is not a block-special file
- the file reached by *<dirnam>* is not a directory
- the block-special file reached by *<spcnam>* is already mounted
- the directory reached by *<dirnam>* already has a block-special file mounted onto it
- the mount table of the operating system is full
- the current effective user is not the system manager

The **mount** function also fails if the media associated with the block-special file was unmounted incorrectly, the media can not be read, the disk does not appear to be a Tektronix 4400 disk, or *<rwflag>* is zero and the disk is write protected.

There is a block device associated with a block-special file. After mounting the block-special file, references to the directory where the file has been mounted now reference the root-directory of the media contained within that block device. If the file is being mounted permitting reading and writing access, **mount** sets an indicator on the media in the device associated with the file, indicating that the media is currently mounted. The **umount()** function clears this indicator.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EBUSY | The operating system´s mount table is full or a device is already mounted on the specified directory *<dirnam>* |
| EDIRTY | The specified file *<spcnam>* was not properly unmounted and could be corrupt |
| EEXIST | The specified file *<spcnam>* is already mounted |
| EIO | The operating-system can not read the data on the device associated with the block-special file specified by *<spcnam>* |
| EMSDR | The path can not be followed for *<spcnam>* or *<dirnam>* |
| ENOENT | There is no entry in the file-system for *<spcnam>* or *<dirnam>* |
| ENOTDIR | The specified file *<dirnam>* is not a directory or the paths in *spcnam* or *<dirnam>* contain a file that is not a directory |
| ENOTBLK | The specified file *<spcnam>* is not a block-special file |
| EWRITPROT | The specified file *<spcnam>* is write protected |

## NOTES

If this function reports EIO or EDIRTY errors, use */etc/diskrepair* to try to salvage the data on the media that could not be mounted.

The **mount** function reports an EIO error if there is no media in the device associated with the block-special file, or if that media is not formatted correctly.

Diskettes written by the **backup** command can not be mounted.

## SEE ALSO

C Library: *addmount(), rmvmount()*

System Call: *mknod(), umount()*

Command: **backup, /etc/diskrepair, /etc/mount, /etc/unmount**

# nice

Change the scheduling priority of a task.

## SYNOPSIS

```
#include <errno.h>
int nice(incr)
     int  incr;
```

### Arguments

| | |
|---|---|
| <incr> | The value to add to the task´s priority |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **nice** function changes the scheduling priority of a task by adding the signed increment *incr* to the value representing the its priority. The **nice** function permits the increment *<incr>* to be negative if the current effective user is the system manager. The **nice** function returns zero if it successfully changes the scheduling priority of the task, otherwise it returns **-1** with *<errno>* set to the system error code.

The **nice** function fails if the increment *<incr>* is negative and the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The increment *<incr>* is negative and the current effective user is not the system manager |

## NOTES

The **nice** function sets the scheduling priority of the task to the maximum priority if the *<incr>* increment causes the priority to exceed the maximum. Likewise, **nice** sets the priority to the minimum priority if the increment causes the it to be less than the minimum.

The higher the value representing the task´s priority, the lower the task´s priority.

## SEE ALSO

Command: **nice**

## open

Open an existing file.

## SYNOPSIS

```
#include <errno.h>
#include <sys/fcntl.h>
int open(pathnam, mode)
     char     *pathnam;
     int       mode;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname to the file to open |
| <mode> | A value describing the requested access permissions |

### Returns

If successful, the file descriptor of the opened file, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **open** function opens the file reached by the pathname in the character-string referenced by *<pathnam>*, sets up access permissions described by the value *<mode>*, and sets the current file position to the beginning of the file. If **open** succeeds, it returns a file descriptor that references the open file. Other functions use this descriptor to reference the file opened by this function, such as **read()**, **write()**, **lrec()**, and **fstat()**, which manipulate open files and their data. If **open** fails, it returns **-1** with *<errno>* set to the system error code.

The **open** function fails if the pathname can not be followed, the path contains a file that is not a directory, the pathname does not exist, the file does not grant the requested access permission to the current effective user, the task has the maximum number of files open, or the requested access permissions are invalid.

The *<mode>* value describes to **open** the requested access permissions. If *<mode>* is O_RDONLY, the **open** function opens the file for reading access. If *<mode>* is O_WRONLY, the **open** function opens the file for writing access. If *<mode>* is O_RDWR, the **open** function opens the file for both reading and writing access. The include-file *<sys/fcntl.h>* contains definitions for the constants O_RDONLY, O_WRONLY, and O_RDWR.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The permissions of the file do not grant the requested access type |
| EINVAL | The *mode* value is invalid |
| EMFILE | The maximum number of files are open |
| EMSDR | The function could not follow the path to the file |
| ENOENT | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

A file descriptor is a non-negative integer that the operating system uses to reference an open file. It is an index into the open file table of the operating system.

## SEE ALSO

C Library: *fclose(), fdopen(), fopen(), freopen()*

System Call: *close(), dup(), dup2(), fstat(), lrec(), pipe(), read(), write()*

# opendir

Open a directory.

## SYNOPSIS

```
#include <sys/dir.h>
DIR *opendir(path)
     char     *path;
```

### Arguments

| | |
|---|---|
| <path> | The address of a character-string containing the pathname of the directory to open |

### Returns

If successful, the address of a directory-stream descriptor to the opened directory, otherwise *(DIR *) NULL*

## DESCRIPTION

The **opendir** function opens for reading the directory reached by the pathname *<path>*, then it attaches the opened directory to a directory-stream. If **opendir** succeeds, it returns a reference to the directory-stream where it has attached the open directory.

If **opendir** fails, it returns *(DIR *) NULL.* The **opendir** function fails if the operating system reports an error. The global variable *<errno>* contains the system error code.

## NOTES

The include-file *<sys/dir.h>* contains definitions for the data types, structures, constants, and functions needed to read directories.

## SEE ALSO

C Library: *closedir(), readdir(), rewinddir(), seekdir(), telldir()*

# pause

Suspend the current task.

## SYNOPSIS

```
#include <errno.h>
int pause()
```

### Arguments

None

### Returns

The **pause** function always returns **-1** with *<errno>* set to the errorcode EINTR.

## DESCRIPTION

The **pause** function suspends the current task indefinitely. This function returns only if the task receives a signal, catches that signal and returns from the function handling that signal, either explicitly using the return statement or implicitly by falling off the end of the function. **Pause** does not return if the task receives a signal that causes the task to terminate. Signals that the task ignores do not affect this function.

If **pause** returns, it always returns **-1** as its result with *<errno>* set to the system error code EINTR.

## ERRORS REPORTED

| | |
|---|---|
| EINTR | The task received a signal, causing it to resume execution |

## SEE ALSO

C Library: *sleep()*

System Call: *alarm(), kill(), signal(), sleep()*

Command: **wait**

# pclose

Close a stream connected to a pipe.

## SYNOPSIS

```
#include <stdio.h>
int pclose(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to close |

### Returns

The termination status of the task at the other end of the pipe if successful, **-1** otherwise

## DESCRIPTION

This function closes the standard I/O stream *<stream>* and frees any resources which were automatically allocated to the stream. If the stream is opened for writing and is buffered, it flushes any buffered data to the associated pipe. After closing the stream, the function waits for the task at the other end of the pipe to terminate.

The function returns **-1** if it encounters an error while closing the stream or the stream was not created by the **popen** function, otherwise it returns the termination status of the task at the other end of the pipe.

## NOTES

This function returns a **-1** if the task at the other end of the pipe does not exist.

This function waits for the specific task at the other end of the pipe. If it receives the termination status for a different task, it discards the status and waits again for the proper task.

## SEE ALSO

C Library: *fflush(), popen(),*

System Call: *close(), pipe()*

## perror

Write a message explaining the error code in *<errno>*.

## SYNOPSIS

```
void perror(ptr)
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of a character-string to write before writing the message describing the error code in *<errno>*, or *(char *) NULL* if none |

### Returns

Void

## DESCRIPTION

If *<ptr>* is not *(char *) NULL*, the **perror** function writes the character-string referenced by *<ptr>* to the standard error I/O stream *<stderr>*, followed by a ':' and a ' '. The **perror** function then writes the error message associated with the value in the variable *<errno>* to *<stderr>*, followed by an end-of-line character.

If *<errno>* is greater than or equal to zero but less than 512, **perror** gets the error message from the */gen/errors/system* file. If *<errno>* is greater than or equal to 512 but less than 1024, it gets the message from the */gen/errors/local* file using (errno % 512) as the error number. If *<errno>* is greater than or equal to 1024, **perror** gets the error message from a file with a name in the form */gen/errors/errorfile%4.4u*, where *%4.4u* is replaced by ((*<errno>*-1024) / 256), using (<errno> % 256) as the error number.

## NOTES

The **perror** function writes the message

```
        No message for errno =
```

followed by the value of *<errno>* if it could not find a message for the current value of *<errno>*. The **perror** function initializes the global variable *sys_nerr* and the global table *<sys_errlist>*.

## SEE ALSO

C Library: *errno, _ierrmsg(), sys_errlist, sys_nerr*

## pffinit

Guarantee that the **cc** command loads the versions of standard I/O functions that contain floating-point conversions.

## SYNOPSIS

```
void pffinit();
```

### Arguments

None

### Returns

Void

## DESCRIPTION

The **pffinit** function guarantees that the **cc** command loads the versions of **fprintf()**, **fscanf()**, **printf()**, **scanf()**, **sprintf()**, and **sscanf()** that contain floating-point conversions.

The **cc** command loads the versions of these functions that contain floating-point conversions only if the C source contains a reference to a floating-point data type. Otherwise, it loads the version of these functions that contains no floating-point conversions.

Symbol name clashes can occur when a user builds libraries containing floating-point data types or references to **pffinit()** and references these routines from a program which contains neither floating-point data types nor a call to **pffinit()**. A user who wishes to print floating-point data types from a program that contains no references to floating-point data types should call **pffinit()** to avoid having the loader bring in both sets of entry points.

## SEE ALSO

C Library: *fprintf(), fscanf(), printf(), scanf(), sprintf(), sscanf()*

Command: **cc**

## phys

Access or release a system resource.

## SYNOPSIS

```
#include <errno.h>
char *phys(code)
     int       code;
```

### Arguments

| | |
|---|---|
| <code> | A value identifying the resource |

### Returns

If successful in accessing a resource, **phys** returns the logical address of the memory associated with the resource; if successful in releasing a resource, **phys** returns *(char *)* **-1**; otherwise it returns *(char *) NULL*

## DESCRIPTION

If the value *<code>* is greater than zero, the **phys** function accesses the resource identified by that value. If **phys** successfully accesses the requested resource, it returns the logical address of the memory mapped for that resource as its result. Otherwise, **phys** returns *(char *) NULL* as its result.

If the value *<code>* is less than zero, the **phys** function releases the resource identified by the absolute value of *<code>* and returns as its result *(char *)* **-1**.

If the value *<code>* is zero, **phys** releases all of the resources allocated by the current task and returns *(char *)* **-1**.

The resources that **phys** makes available depend on the particular implementation of the operating system, so the meaning of the value *<code>* differs from implementation to implementation. The following table describes the meaning of the absolute value of *<code>* for the Tektronix 4404 Series:

| | |
|---|---|
| 1 | The 128K bit-map |
| 2 | The first shared 4K page |
| 3 | The second shared 4K page |
| 4 | The time-of-day clock |

## ERRORS REPORTED

| | |
|---|---|
| EINVAL | The value <*code*> is out of range |

## NOTES

The *phys* function ignores requests to release resources that have not been allocated to the task. Likewise, **phys** ignores requests to allocate resources that are already allocated by the task.

## pipe

Create a pipe.

## SYNOPSIS

```
#include <errno.h>
int pipe(fds)
     int      (*fds)[2];
```

### Arguments

| | |
|---|---|
| <fds> | The address of a two-element array of integers to receive the pipe's input and output file descriptors |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **pipe** function creates a pipe, which is a first-in, first-out I/O mechanism typically used to send data from one task to another. It saves the pipe's input file descriptor in the first element of the array referenced by *<fds>* and it saves the pipe's output file descriptor in the second element of that array. The function returns zero if it successfully creates a pipe, otherwise **pipe** returns **-1** with *<errno>* set to the system error code.

The **pipe** function fails if the task has more than two less than the maximum number of files the system permits a task to have open.

Reading from a pipe whose buffers are not full and whose input file descriptor has not been closed suspends the task until the pipe is filled or the pipe's output file descriptor is closed. Writing to a pipe whose buffers are full suspends the task until all of the data written to the pipe has been read.

Reading from a pipe whose buffers contain no data and whose output file descriptor is closed causes the function attempting to read data from the pipe to report an end-of-file error. Writing to a pipe whose input file descriptor has been closed causes the function attempting to write the data to the pipe to report a broken pipe error.

Typically, a task creates a pipe using this function, the task then executes a **fork()**, duplicating the pipe's input and output file descriptors for the child (created) task. The sending task (the task that is to send data through the pipe) closes the input file descriptor of the pipe and writes data to the output file descriptor of the pipe. The receiving task (the task that receives data from the pipe) closes the output file descriptor of the pipe and reads data from the input file descriptor of the pipe.

## ERRORS REPORTED

| | |
|---|---|
| EMFILE | The task has too many files open to create a pipe |

## NOTES

Undefined behavior results if a task attempts to use both the input file descriptor and the output file descriptor.

## SEE ALSO

System Call: *close(), fstat(), open(), read(), write()*

Command: **shell**

# popen

Open a pipe and attach it to a standard I/O stream.

## SYNOPSIS

```
#include <stdio.h>
FILE *popen(command, mode)
     char      *command;
     char      *mode;
```

### Arguments

| | |
|---|---|
| <command> | The address of a character-string containing a command to which the pipe is connected |
| <mode> | The address of a character-string containing the open mode |

### Returns

If successful, the stream to which the pipe has been attached, otherwise **(FILE *) NULL**

## DESCRIPTION

This function opens a pipe between the calling program and the command in the character-string referenced by *<pathnam>*. The character-string referenced by *<mode>* describes to the function the access type desired by the calling program. The function then attaches the pipe to a standard I/O stream.

If the function succeeds, it returns the standard I/O stream as its result. Otherwise, it returns **(FILE *) NULL**. The function fails if the operating system reports an error, the program has the maximum number of streams open, the calling program cannot create a task to execute the command, or the open mode is not valid. If the operating system reports an error, **errno** will contain the system error code.

The open mode describes the type of access requested for the pipe by the calling program. Valid open modes are **r**, and **w**, for read, or write access, respectively.

If the open mode is **r**, the function opens the stream for reading. The pipe is connected to the standard output descriptor of the command.

If the open mode is **w**, the function opens the stream for writing. The pipe is connected to the standard input descriptor of the command.

## NOTES

The include-file <*stdio.h*> defines the data type **FILE**. This data type is a structure containing all of the information about an open stream.

For brevity, this and other manual pages discuss a pointer to the data type **FILE** as simply a **stream**, instead of calling it a pointer to a structure defining the characteristics of a stream.

A stream created by the **popen** function should be closed by the **pclose** function.

The calling program will receive an **end-of-file** response if it should read the stream after the command has terminated or closed the pipe.

If the calling program should write to the stream after the command has closed the pipe or terminated, then the calling program will receive a **broken pipe** signal.
**pr**, the piece of junk, is losing the
The following .in -0 is an attempt to remind it
to do the indentation correctly.

## SEE ALSO

C Library: *fgetc(), fgets(), fputc(), fputs(), fread(), fwrite(), pclose()*

System Call: *close(), pipe(), signal()*

## pow

Raise a value to a power.

## SYNOPSIS

```
#include <math.h>
double pow(x, y)
     double    x;
     double    y;
```

### Arguments

| | |
|---|---|
| <x> | The value to raise |
| <y> | The power to raise *<x>* to |

### Returns

The value *<x>* raised to the power *<y>*

## DESCRIPTION

The **pow** function calculates the value of *<x>* raised to the power *<y>*. **Pow** returns the calculated value as the result.

If *<x>* is greater than zero, **pow** permits any value of *<y>*. If *<x>* is zero, **pow** permits *<y>* to be any non-zero value, otherwise it reports a singularity error. If *<x>* is less than zero, **pow** permits *<y>* to be value that is an integer, otherwise it reports a domain error. If the magnitude of the result is larger than that what can be represented by a *double*, the **pow** function reports an overflow error.

If **pow** detects a singularity error, it calls **matherr()** passing to it the address of a filled *<struct>* exception structure. It sets the element *<type>* to SING, *<name>* to the address of the character string *<pow>*, *<arg1>* to *<x>*, and *<arg2>* to *<y>*. If **matherr()** returns 0, **pow** writes the message

```
pow() error:  Both arguments are 0.0
```

to the standard error I/O stream *<stderr>* and sets *<errno>* to EDOM. The return value, which is system-dependent, can be found in the manual page for **kill()**. If **matherr()** returns a value other than 0, **pow** returns as its result the value *retval* found in the *<struct>* exception structure whose address was passed to **matherr()**.

If **pow** detects a domain error, it calls **matherr()** passing to it the address of a filled *<struct>* exception structure. It sets the element *<type>* to DOMAIN, *<name>* to the address of the character string *<pow>*, *<arg1>* to *<x>*, and *<arg2> to <y>*. *If* **matherr()** *returns 0,* **pow** *writes the message*

```
        pow() error:  Negative base with non-integer power
```

to the standard error I/O stream *<stderr>*, sets *<errno>* to EDOM, and returns 0.0 as its result. Otherwise, **pow** returns the value *retval* found in the *<struct>* exception structure whose address was passed to **matherr()**.

If **pow** detects a overflow error it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. It sets the element *<type>* to OVERFLOW, *<name>* to the address of the character string *<pow>*, *<arg1>* to *<x>*, and *<arg2>* to *<y>*. If **matherr()** returns 0, **pow** sets *<errno>* to ERANGE and returns a value whose magnitude is the largest value representable by the data type *double* (HUGE), signed as the result would have been signed had it not been larger than can be represented by that data type. Otherwise, it returns as its result the value *retval* found in the *<struct>* exception structure whose address was passed to **matherr()**.

## SEE ALSO

C Library: *matherr()*

# printf

Write formatted data to *stdout.*

## SYNOPSIS

```
#include <stdio.h>
int printf(format [,arglist])
     char     *format;
```

### Arguments

| | |
|---|---|
| <format> | The address of a character-string containing a format description |

### Returns

The number of characters written to *<stdout>* or EOF if an error occurred

## DESCRIPTION

The **printf** function generates characters from the format description in the character-string referenced by *<format>* and the arguments in the argument-list *<arglist>*, if any, and writes these characters to the standard I/O output stream *<stdout>*. It returns as its result the number of characters written to *<stdout>*.

The format description in the character-string referenced by *<format>* contains literal characters and field descriptions. The **printf** function writes literal characters to *<stdout>* with no interpretation. The **printf** function interprets field descriptions to determine what characters it generates, what type of argument it consumes, if any, from the argument list *<arglist>*, and the type of conversion it performs. The number of arguments and the type of the arguments in the argument list *<arglist>* depends on the format description. The argument list can be omitted.

For a complete description of the *<format>* argument, see the manual page for **fprintf()**.

## NOTES

The **printf** function writes characters to *<stdout>* using **fputc()**. If *<stdout>* is buffered, standard I/O does not write characters to the file attached to the stream until it fills the stream's buffer or closes the stream. If *<stdout>* is line-buffered (buffered and attached to a file that is a terminal), standard I/O does not write characters to the file attached to the stream until it fills the stream's buffer, closes the stream, writes an end-of-line character (EOL) to the stream, or reads data from a terminal.

The include-file *<stdio.h>* defines the functions and constants available in standard I/O. This file must be included in the C source before the first reference to this function.

The C library contains two versions of the **fprintf()** function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the function **pffinit()**. Otherwise, it loads the version which contains no floating-point conversions.

## SEE ALSO

C Library: *ecvt(), fcvt(), fdopen(), fopen(), fprintf(), fputc(), fscanf(), gcvt(), pffinit(), scanf(), sprintf(), sscanf(), stdout*

Command: **cc**

# profil

Start or stop monitoring the current task.

## SYNOPSIS

```
int profil(bufad, bufsiz, lowpc, scale)
     char     *bufad;
     int       bufsiz;
     int       lowpc;
     int       scale;
```

### Arguments

| | |
|---|---|
| <bufad> | The address of the buffer to contain monitoring information |
| <bufsiz> | The number of bytes in the buffer whose address is <bufad> |
| <scale> | A value indicating the monitoring granularity |
| <lowpc> | The lowest address to monitor in the task |

### Returns

Zero

## DESCRIPTION

If *<scale>* is not 0 or 1, the **profil** function requests that the operating system begin monitoring the current task, using the buffer whose address is *<bufad>* and contains *<bufsiz>* bytes, as the monitor buffer, beginning at the program address *<lowpc>*, with a granularity of *<scale>*. If *<scale>* is 0 or 1, **profil** requests that the operating system stop monitoring the current task. The **profil** function always returns zero as its result.

While monitoring a task, the operating system examines the task at each tick on the system clock, which occurs every tenth of a second. It takes the current program counter of the task, subtracts from it the value *<lowpc>*, divides the result by *<scale>*, then multiplies the quotient by two. If the product is less than the value *<bufsiz>*, it adds the product to the address of the buffer *<bufad>*, then increments the word at that resulting address by one.

## ERRORS REPORTED

None

## NOTES

The buffer containing the values incremented at each clock tick must begin at an even address.

The operating system's monitoring mechanism only uses the least-significant byte of the <*scale*> argument. After checking for 0 or 1, if <*scale*> is not a power of 2, it rounds the value up to the next power of 2 and uses that value as the scaling factor.

The argument <*lowpc*> is an address that has been cast into an *int*.

The operating system automatically stops monitoring a task when that task calls the **exec()** function.

The operating system does not automatically stop monitoring a task when that task calls the *fork()* function.

## SEE ALSO

C Library: *monitor()*

System Call: *exec(), fork()*

Command: **cc**

## putc

Write a character to a stream.

## SYNOPSIS

```
#include <stdio.h>
int putc(c, stream)
     char      c;
     FILE      *stream;
```

### Arguments

| | |
|---|---|
| <c> | The character to write |
| <stream> | The standard I/O stream to write to |

### Returns

The value written if successful, EOF otherwise

## DESCRIPTION

The **putc** function writes the character *<c>* to the standard I/O stream *<stream>*. The **putc** function returns the character written as its result if it successfully writes the character to the stream, otherwise it returns EOF.

## NOTES

If the stream is buffered, standard I/O flushes the buffered data whenever the buffer fills or the stream closes.

If the stream is line-buffered (buffered and attached to a character-special device), standard I/O flushes the buffered data whenever the buffer fills, the stream closes, a standard I/O function writes an EOL character to the stream, or a standard I/O function reads data from a character-special device (a terminal).

## SEE ALSO

C Library: *fdopen(), fopen(), fputc(), getc(), putchar()*

# putchar

Write a character to *<stdout>*.

## SYNOPSIS

```
#include <stdio.h>
int putchar(c)
     char      c;
```

### Arguments

| | |
|---|---|
| c | The character to write |

### Returns

The value written if successful, EOF otherwise.

## DESCRIPTION

The **putchar** function writes the character *<c>* to the standard I/O standard output stream *<stdout>*. The **putchar** function returns the character written as its result if it successfully wrote the character to *<stdout>*, otherwise, it returns EOF.

## NOTES

If the stream is buffered, standard I/O flushes the buffered data whenever the buffer fills or the stream closes.

If the stream is line-buffered (buffered and attached to a character-special device), standard I/O flushes the buffered data whenever the buffer fills, the stream closes, a standard I/O function writes an EOL character to the stream, or a standard I/O function reads data from a character-special device (a terminal).

## SEE ALSO

C Library: *fdopen(), fopen(), fputc(), getchar(), putc(), stdout*

# putenv

Modify or add an environment-variable definition to the environment list.

## SYNOPSIS

```
int putenv(ptr)
     char     *ptr;
```

### Arguments

| | |
|---|---|
| <ptr> | The address of a character-string containing the environment-variable definition |

### Returns

Zero if the function was unable to obtain enough memory (using **malloc()**) to relocate the environment list, non-zero otherwise ((**char** *) NULL if the name was not found in the list

## DESCRIPTION

This function changes an existing environment-variable definition or adds a new definition to the environment list. If the definition in the character-string referenced by *<ptr>* defines a variable that already exists in the environment list, this function changes the definition of that environment variable. Otherwise, it expands the environment list by appending the definition found in the character-string referenced by *<ptr>*. If expanding the environment list requires that the list be moved, this function updates the external variable **environ** so that it references the expanded list.

If the function is unable to obtain enough main memory for the expanded environment list, it returns zero as its result, otherwise it returns a non-zero value.

## NOTES

The environment list is a variable-length array of addresses terminated by the null-address. Each address references a character-string defining a variable of the current environment. Each character-string is of the form *<name>=<value>* where *<name>* is the name of the environment variable and *<value>* is the definition of that variable. The function does nothing if the character-string referenced by *<ptr>* is not of the form *<name>=<value>*.

The updated environment list will contain a reference to the character-string referenced by *<ptr>*. Modifying that string after calling this function will result in redefinition of the environment list.

The character-string referenced by *<ptr>* should be in static memory because the definition of the environment variable may exist after the scope which called this function is exited.

Under no circumstances does this function modify the third argument to the main procedure main().

## SEE ALSO

C Library: *environ, getenv()*

# put_FPU_control

Change the contents of the MC68881 control and status registers

## SYNOPSIS

```
#include <float_interrupt.h>
void put_FPU_control(buffer)
     struct FPU_control *buffer;
```

### Arguments

| | |
|---|---|
| <buffer> | The address of the structure which contains the updated contents of the MC68881 registers |

### Returns

None

## DESCRIPTION

The **put_FPU_control** function changes the contents of the MC68881 control and status registers (FPCR and FPSR, respectively).

This function expects *<buffer>* to be the address of a structure defined as:

```
struct FPU_control {
    struct control_register fpcr;   /* control register */
    struct status_register fpsr;    /* status register */
};
```

The organization of the individual registers is defined by these structures:

```
struct control_register {
    unsigned         :4;     /* unused */
    unsigned rnd     :2;     /* rounding mode */
    unsigned prec    :2;     /* rounding precision */
    unsigned inex1   :1;     /* inexact decimal input */
    unsigned inex2   :1;     /* inexact operation */
    unsigned dz      :1;     /* divide by zero */
    unsigned unfl    :1;     /* underflow */
    unsigned ovfl    :1;     /* overflow */
    unsigned operr   :1;     /* operand error */
    unsigned snan    :1;     /* signaling NAN */
    unsigned bsun    :1;     /* branch/set on unordered */
    unsigned         :16;    /* unused */
};

struct status_register {
    unsigned         :3;     /* unused */
    unsigned inex    :1;     /* accrued inexact */
    unsigned adz     :1;     /* accrued divide-by-zero */
    unsigned aunfl   :1;     /* accrued underflow */
    unsigned aovfl   :1;     /* accrued overflow */
    unsigned iop     :1;     /* invalid operation */
    unsigned inex1   :1;     /* inexact decimal input */
    unsigned inex2   :1;     /* inexact operation */
    unsigned dz      :1;     /* divide by zero */
    unsigned unfl    :1;     /* underflow */
    unsigned ovfl    :1;     /* overflow */
    unsigned operr   :1;     /* operand error */
    unsigned snan    :1;     /* signaling NAN */
    unsigned bsun    :1;     /* branch/set on unordered */
    unsigned quotient :7;    /* 7 least significant bits of quotient
    unsigned s       :1;     /* sign of quotient */
    unsigned nan     :1;     /* not a number or unordered */
    unsigned i       :1;     /* infinity */
    unsigned z       :1;     /* zero */
    unsigned n       :1;     /* negative */
    unsigned         :4;     /* unused */
};
```

For a description of the fields in the control and status registers, refer to the MC68881 hardware manual.

## NOTES

The include-file *<float_interrupt.h>* contains the definitions of the above structures.

## SEE ALSO

C Library: *get_FPU_control()*

System Call: *FPU_resume(), get_FPU_exception(), put_FPU_exception()*

## put_FPU_exception

Update MC68881 coprocessor exception-information

## SYNOPSIS

```
#include <errno.h>
#include <float_interrupt.h>
int put_FPU_exception(buffer)
     struct FPU_interrupt_data *buffer;
```

### Arguments

| | |
|---|---|
| <buffer> | The address of a buffer that contains the updated MC68881 coprocessor exception-information |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **put_FPU_exception** function updates the exception information provided by the MC68881 coprocessor when it detected an error for which it is generating interrupts. This function is intended for use in an interrupt-handling routine when attempting to recover from errors detected by the MC68881 coprocessor. The user may modify selected portions of the exception information in an attempt to recover from the error. After making the desired modifications, the user calls this function to transfer the changes to the CPU and MC68881 exception stack frames.

The **put_FPU_exception** function returns zero if it successfully updates the exception stack frames, otherwise, it returns **-1** with *<errno>* set to the system error code.

This function expects *<buffer>* to be the address of a structure that is defined as:

```
struct FPU_interrupt_data {
    struct state_frame FPU_frame;     /* FPU state frame */
    struct control_register fpcr;     /* control register */
    struct status_register fpsr;      /* status register */
    short *fpiar;                     /* instruction address regist
    fpreg fp[8];                      /* floating-point data regist
    long CPU_data_register[8];        /* CPU "D" registers */
    long CPU_address_register[8];     /* CPU "A" registers */
    struct exception_frame CPU_frame;    /* CPU exception frame */
};
```

The individual components of this structure are defined as:

```
struct exception_frame {
    unsigned short sr;                /* CPU status register */
             short *CPU_pc;           /* CPU program counter */
             short frame_type;        /* exception frame type */
             short *pc;               /* program counter */
    unsigned short ir;                /* internal register */
    unsigned short operation;         /* operation word */
             short *address;          /* effective address */
};

struct control_register {
    unsigned        :4;       /* unused */
    unsigned rnd    :2;       /* rounding mode */
    unsigned prec   :2;       /* rounding precision */
    unsigned inex1  :1;       /* inexact decimal input */
    unsigned inex2  :1;       /* inexact operation */
    unsigned dz     :1;       /* divide by zero */
    unsigned unfl   :1;       /* underflow */
    unsigned ovfl   :1;       /* overflow */
    unsigned operr  :1;       /* operand error */
    unsigned snan   :1;       /* signaling NAN */
    unsigned bsun   :1;       /* branch/set on unordered */
    unsigned        :16;      /* unused */
};
```

```
struct status_register {
    unsigned          :3;      /* unused */
    unsigned inex     :1;      /* accrued inexact */
    unsigned adz      :1;      /* accrued divide-by-zero */
    unsigned aunfl    :1;      /* accrued underflow */
    unsigned aovfl    :1;      /* accrued overflow */
    unsigned iop      :1;      /* invalid operation */
    unsigned inex1    :1;      /* inexact decimal input */
    unsigned inex2    :1;      /* inexact operation */
    unsigned dz       :1;      /* divide by zero */
    unsigned unfl     :1;      /* underflow */
    unsigned ovfl     :1;      /* overflow */
    unsigned operr    :1;      /* operand error */
    unsigned snan     :1;      /* signaling NAN */
    unsigned bsun     :1;      /* branch/set on unordered */
    unsigned quotient :7;      /* 7 least significant bits of quotient
    unsigned s        :1;      /* sign of quotient */
    unsigned nan      :1;      /* not a number or unordered */
    unsigned i        :1;      /* infinity */
    unsigned z        :1;      /* zero */
    unsigned n        :1;      /* negative */
    unsigned          :4;      /* unused */
};

typedef unsigned char fpreg[12]; /* one floating point register */

struct exception_frame {
    unsigned short sr;                /* CPU status register */
             short *CPU_pc;           /* CPU program counter */
             short frame_type;        /* exception frame type */
             short *pc;               /* program counter */
    unsigned short ir;                /* internal register */
    unsigned short operation;         /* operation word */
             short *address;          /* effective address */
};
```

The user may not modify the *state_frame* and *exception_frame* structures. The contents of the other structures are copied into the CPU and coprocessor stack frames. For interpretation of the information contained in these structures, consult the appropriate hardware manuals.

## ERRORS REPORTED

| | |
|---|---|
| EBDCL | The CPU cannot support the MC68881 coprocessor |
| ENOFPUDATA | There is no exception information to be updated |

## NOTES

Exception information is available only after the MC68881 interrupts the CPU. The user must have previously enabled these interrupts by setting the appropriate bits in the MC68881 control register.

The user must call the **FPU_resume**() routine to resume execution of the interrupted MC68881 instruction and exit the interrupt-handling routine. Attempting to exit the interrupt-handling routine by using the *return* statement may lead to unpredictable results because the program counter stored on the stack may not be correct.

The include-file *<float_interrupt.h>* contains the above structure definitions.

## SEE ALSO

C Library: *get_FPU_control(), put_FPU_control()*

System Calls: *FPU_resume(), get_FPU_exception()*

# putpwent

Format and write a system password-file record.

## SYNOPSIS

```
#include <pwd.h>
int putpwent(ptr, stream)
     struct passwd *ptr;
     FILE          *stream;
```

### Arguments

| | |
|---|---|
| <ptr> | Address of a structure containing the information to write |
| <stream> | The standard I/O stream to write to |

### Returns

Zero if the record was successfully written, EOF otherwise.

## DESCRIPTION

The **putpwent** function generates a character-string from the information in the structure referenced by *<ptr>* and writes that character-string to the standard I/O output stream *<stream>*. It generates the character-string in the format required by the system-password file. The **putpwent** function returns zero as its result if it successfully formats and writes the record, otherwise it returns EOF.

The function generates the character-string using the following **sprintf()** format-string:

```
"%s:%s:%d:%s:%s\n"
```

The format of the structure referenced by the result of this function is defined by the include-file *<pwd.h>* and is defined as:

```
struct passwd
{
     char      *pw_name;
     char      *pw_passwd;
     int        pw_uid;
     char      *pw_dir;
     char      *pw_shell;
};
```

The *pw_name* entry is the address of a character-string containing the user-name. *pw_passwd* is the address of a character-string containing the encrypted password. *pw_uid* contains the user's identifying number (user-ID). *pw_dir* is the address of a character-string containing the user's initial home-directory. *pw_shell* is the address of a character-string containing containing the shell-command for the first program to run after logging on.

## NOTES

The **putpwent** function uses standard I/O and enlarges more than expected a program not otherwise using standard I/O.

The *pw_passwd* and *pw_shell* entries in the structure referenced by *<ptr>* may be *(char *) NULL* or they may reference a null-string. In either case, **putpwent** uses a null-string when generating the system password record.

## SEE ALSO

C Library: *endpwent(), getpw(), getpwent(), getpwnam(), getpwuid(), setpwent()*

Command: **password**

# puts

Write a character-string to *stdout.*

## SYNOPSIS

```
#include <stdio.h>
int puts(s)
     char     *s;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to write |

### Returns

Zero if successful, EOF otherwise.

## DESCRIPTION

The **puts** function writes the character-string referenced by <*s*>, followed by an end-of-line character (EOL), to the standard I/O standard output stream <*stdout*>. The **puts** function returns zero if it successfully writes the character-string to <*stdout*>, otherwise, it returns EOF.

## NOTES

The **puts** function does not write the null-character terminating the character-string to <*stdout*>.

## SEE ALSO

C Library: *fdopen(), fopen(), fputs(), gets(), putchar(), stdout*

# putw

Write a word to a stream.

## SYNOPSIS

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

### Arguments

| | |
|---|---|
| <wd> | The value to write |
| <stream> | The standard I/O stream |

### Returns

The value written if successful, EOF otherwise

## DESCRIPTION

The **putw** function casts the *int* argument *<wd>* into a *short* then writes that *short* to the standard I/O output stream referenced by *<stream>*. It writes the high-order byte first, then the low-order byte. It then casts the *short* written into an *int* and returns that value as its result. If it detects an error, it returns EOF as its result.

## NOTES

The value EOF is a valid value to write, so the functions **ferror()** and **feof()** should be used to check for error and end-of-file conditions for the stream.

There are no boundary alignment requirements for writing a word to the stream.

## SEE ALSO

C Library: *fdopen(), fopen(), getw(), putc()*

## qsort

Sort data.

## SYNOPSIS

```
int qsort(base, nr, width, compar)
     char             *base;
     unsigned int      nr;
     unsigned int      width;
     int              (*compar)();
```

### Arguments

| | |
|---|---|
| <base> | The address of the data to sort |
| <nr> | The number of records in the data |
| <width> | The number of bytes in a record of data |
| <compar> | The address of a function to use to compare two records |

### Returns

The **qsort** function returns **0** if it successfully sorted the data, otherwise **qsort** returns **-1**.

## DESCRIPTION

The **qsort** function sorts in place the data at the address *<base>*. The data contain *<nr>* records with each record *<width>* in length. The **qsort** function uses the function whose address is *<compar>* to compare two records of data. **Qsort** returns 0 as its result if it successfully sorted the data, otherwise it returns **-1** as its result.

The function whose address is *<compar>* is a function returning an *int* with two arguments, both addresses of records of data. The **qsort** function returns a value less than, equal to, or greater than zero if the record referenced by the first argument is less than, equal to, or greater than the record referenced by the second argument.

## NOTES

The only possible condition resulting in a result of -1 is a record size <*width*> larger than the maximum. The maximum is currently 256 bytes.

The *base* argument should be a pointer-to-element cast into a *(char *)*.

## SEE ALSO

None

# rand

Generate a random number.

## SYNOPSIS

```
int rand();
```

### Arguments

None

### Returns

A random number

## DESCRIPTION

The **rand** function generates a random number and returns that value as its result. The number is between 0 and 32767 inclusive. This function is a pseudo-random number generator, generating the next number in a sequence. The previous number in the sequence is the seed set automatically at the start of the program, the seed set explicitly by the **rrand()** or **srand()** functions, or the previously generated random number.

## NOTES

The sequence of numbers generated by this function from a particular seed is always reproducible.

## SEE ALSO

C Library: *rrand(), srand()*

# read

Read data from an open file.

## SYNOPSIS

```
#include <errno.h>
int read(fildes, bufad, nbytes)
     int       fildes;
     char     *bufad;
     int       nbytes;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the open file from which to read data |
| <bufad> | The address of a buffer to contain the data read |
| <nbytes> | The maximum number of byte to read |

### Returns

The number of bytes read if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **read** function reads data from the open file referenced by the file descriptor *<fildes>*, beginning at the current file position, reading a maximum of *<nbytes>* bytes, writing the data read into the buffer with the address *<bufad>*. The function reads data until it reads the maximum number of bytes, it reaches the end of the associated file, or, if the associated file is a character-special file (terminal), **read** reads an end-of-line character. If the associated file is one that can be repositioned, the function changes the current file position to that of the data immediately following the last byte read.

If the **read** function successfully reads data, it returns the number of bytes it read as its result. If **read** encounters the end of the file before reading any data, it returns zero as its result. Otherwise, **read** returns **-1** as its result and sets *<errno>* to the system error code describing the error.

The **read** function fails if the file descriptor *<fildes>* is out of range, does not reference an open file, or references an open file that is not open for reading. It also fails if an I/O error occurs while reading data or the requested count *<nbytes>* is negative. The **read** function also fails if the task receives and catches a signal while reading data from a slow device, such as a terminal.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The specified file is not opened for reading |
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EINTR | The task received an caught a signal while the function was reading from a slow device |
| EIO | The operating system reports an I/O error |
| EINVAL | The value <*nbytes*> is not valid or the file descriptor <*fildes*> is out of range |

## NOTES

The data in the buffer may change if the function reports an I/O error.

## SEE ALSO

C Library: *fread()*

System Call: *creat(), dup(), dup2(), open(), pipe(), write()*

# readdir

Read the next entry in an open directory.

## SYNOPSIS

```
#include <sys/dir.h>
struct direct *readdir(pdir)
     DIR      *pdir;
```

### Arguments

| | |
|---|---|
| <pdir> | A reference to the directory-stream to read from |

### Returns

If **readdir** is successful, it returns the address of a structure containing the information in the next entry of the directory, otherwise **readdir** returns *(struct direct *) NULL*

## DESCRIPTION

The **readdir** function attempts to read the next entry from the directory attached to the directory-stream referenced by *<pdir>*. If it succeeds, it decodes the information in that entry, places the decoded information in a structure, and returns as its result the address of that structure.

If **readdir** fails, it returns as its result *(struct direct)* NULL. The **readdir** function fails if the directory-stream reference is not valid or there are no more entries in the directory attached to the directory-stream.

The structure referenced by the result of this function is defined as:

```
struct direct
{
     short     d_ino;
     short     d_namlen;
     char      d_name[MAXNAMLEN+1];
}
```

The *d_ino* element is the file descriptor number in the directory entry. *d_namlen* is the length of the filename in the directory entry. *d_name* is an array containing the null-terminated filename in the directory-entry. The constant MAXNAMLEN is the maximum length of a filename (an element of a pathname).

## NOTES

The include-file *<sys/dir.h>* contains definitions for the data types, structures, constants, and functions needed to read directories.

The **readdir** function skips empty directory entries.

## SEE ALSO

C Library: *closedir(), opendir(), rewinddir(), seekdir(), telldir()*

# realloc

Reallocate an allocated block of data.

## SYNOPSIS

```
char *realloc(buf, size)
     char           *buf;
     unsigned int    size;
```

### Arguments

| | |
|---|---|
| <buf> | The address of the allocated buffer to reallocate |
| <size> | The requested new size of the buffer, in bytes |

### Returns

The address of the reallocated buffer

## DESCRIPTION

The **realloc** function changes the size of the allocated buffer with the address *<buf>* to the *<size>* bytes. If the space allocated to the buffer is large enough to accommodate *<size>* bytes, **realloc** returns from the buffer as much space as possible to the arena of available memory and returns *<buf>* as its result. Otherwise, **realloc** returns *<buf>* to the arena of available memory and allocates a buffer of *<size>* bytes. If successful, **realloc** copies the data in the original buffer to the newly allocated buffer and returns the address of the allocated buffer as its result. Otherwise, **realloc** returns *(char *) NULL* as its result.

## NOTES

The original block is destroyed if the function returns *(char *) NULL.*

## SEE ALSO

C Library: *calloc(), free(), malloc()*{

System Call: *brk(), cdata(), sbrk()*

# rewind

Rewind a stream.

## SYNOPSIS

```
#include <stdio.h>
int rewind(stream)
     FILE     *stream;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream to rewind |

### Returns

Void

## DESCRIPTION

The **rewind** function rewinds the stream referenced by *<stream>*. Rewinding a stream positions the stream to the beginning of its attached file.

## NOTES

**Rewind** undoes the effect of an **ungetc()** function.

**Rewind** does not rewind a stream that is opened in *append* mode.

**Rewind** does not rewind a stream that is attached to a character-special file (terminal).

## SEE ALSO

C Library: *fdopen(), fopen(), fseek(), ftell()*

System Call: *lseek()*

# rewinddir

Rewind a directory-stream.

## SYNOPSIS

```
#include <sys/dir.h>
void rewinddir(pdir)
     DIR      *pdir;
```

### Arguments

| | |
|---|---|
| <pdir> | A reference to a directory-stream |

### Returns

Void

## DESCRIPTION

The **rewinddir** function rewinds the directory-stream referenced by *<pdir>*. The next read-operation requested on the directory-stream *<pdir>* reads the first entry in the directory.

## NOTES

The include-file *<sys/dir.h>* contains definitions for the data types, structures, constants, and functions needed to read directories.

## SEE ALSO

C Library: *closedir(), opendir(), readdir(), seekdir(), telldir()*

## rindex

Find the last occurrence of a character in a character-string.

## SYNOPSIS

```
char *rindex(s, c)
     char     *s;
     char      c;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to search |
| <c> | The character to search for |

### Returns

The address of the last occurrence of the character in the string, or *(char *) NULL if the string does not contain the character*

## DESCRIPTION

The **rindex** function searches the character-string with the address *<s>* for the last occurrence of the character *<c>*. If the string contains the character, **rindex** returns as its result the address of the last occurrence of the character. Otherwise, it returns *(char *) NULL.*

## NOTES

The **rindex** function is obsolete. It is only included for compatibility with older C libraries. New applications should use **strrchr**().

## SEE ALSO

C Library: *index(), strchr(), strrchr()*

# rmvmount

Remove an entry from the system mount table.

## SYNOPSIS

```
void *addmount(device)
     char     *device;
```

### Arguments

| | |
|---|---|
| <device> | The address of a character-string containing the pathname of the device which is mounted |

### Returns

Void

## DESCRIPTION

This function removes an entry from the system's mount-table file.

## NOTES

The **rmvmount()** function does not perform an actual unmount of the device on the directory; it only manipulates the system's mount-table file.

## SEE ALSO

C Library: *addmount()*

System Call: *mount(), umount()*

Command: */etc/mount, /etc/unmount*

## rrand

Set the seed of the random number generator to a value generated from the current system-time value.

## SYNOPSIS

```
void rrand(seed);
```

### Arguments

None

### Returns

Void

## DESCRIPTION

The **rrand** function sets the seed of the pseudo-random number generator to a value generated from current system-time value. The value generated is that of the low 15 bits of the system-time value.

## NOTES

The system-time value is the current time expressed in the number of seconds since the epoch. The system defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

The seed is the value from which the next random number is generated.

The random number generating function **rand()** always generates the same sequence of random numbers from a particular seed.

## SEE ALSO

C Library: *rand(), srand()*

System Call: *time()*

## rump_create

Create a new managed resource.

## SYNOPSIS

```
#include <errno.h>
int rump_create(resource)
    char *resource
```

### Arguments

<resource> The name of the resource

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **rump_create** function creates a new named resource. The purpose of such resources is to provide a mechanism for controlling access to physical resources such as I/O devices or special shared memory. There are four operations that may be applied to a named resource. These are:

| | |
|---|---|
| create — | create the resource |
| destroy — | remove the resource from the system |
| enqueue — | obtain exclusive access to the resource |
| dequeue — | relinquish access to the resource. |

If the **rump_create** function succeeds, a new named resource is created with the name given by the argument *<resource>*. This must be a NULL terminated character string of 16 or fewer characters (including the NULL). Otherwise, **rump_create** returns **-1** with *<errno>* set to the system error code.

The **create** function does not give access of the resource to to the creator.

## ERRORS REPORTED

| | |
|---|---|
| EEXIST | The named resource already exists |
| ENOSPC | The maximum number of resources already exists |

## NOTES

A resource name is a NULL terminated string of no more than 16 characters.

## SEE ALSO

C Library: *rump_destroy(), rump_enqueue(), rump_dequeue()*

# rump_dequeue

relinquish access to a named resource.

## SYNOPSIS

```
#include <errno.h>
int rump_enqueue(resource)
    char *resource
```

### Arguments

| | |
|---|---|
| <resource> | The name of the resource |

### Returns

Zero if successful, otherwise **-1** with <*errno*> set to the system error code

## DESCRIPTION

The **rump_dequeue** function releases access to the named resource.

If the **rump_dequeue** function succeeds, access to the named resource with the name given by the argument <*resource*> is given up. Otherwise, **rump_dequeue** returns **-1** with <*errno*> set to the system error code.

If any other tasks are currently waiting for access to the resource, then the first such task is given access to the resource.

## ERRORS REPORTED

| | |
|---|---|
| ENOENT | The named resource does not exist |
| EBADF | The task does not currently have access to the resource |

## NOTES

A resource name is a NULL terminated string of no more than 16 characters.

## SEE ALSO

C Library: *rump_create(), rump_destroy(), rump_enqueue()*

# rump_destroy

Destroy a managed resource.

## SYNOPSIS

```
#include <errno.h>
int rump_destroy(resource)
    char *resource
```

### Argument

<resource> The name of the resource

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **rump_destroy** function destroys a named resource. If the function succeeds, the named resource with the name given by the argument *<resource>* is destroyed. Otherwise, **rump_destroy** returns **-1** with *<errno>* set to the system error code.

Only an idle resource can be destroyed. If any task currently has access to the resource, the destroy function is not permitted.

## ERRORS REPORTED

| | |
|---|---|
| ENOENT | The named resource does not exist |
| EBUSY | The resource is currently busy |

## NOTES

A resource name is a NULL terminated string of no more than 16 characters.

## SEE ALSO

C Library: *rump_create(), rump_enqueue(), rump_dequeue()*

## rump_enqueue

Obtain exclusive access to a named resource.

## SYNOPSIS

```
#include <errno.h>
int rump_enqueue(resource)
    char *resource
```

### Arguments

<resource> The name of the resource

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **rump_enqueue** function obtains exclusive access to the named resource for the task.

If the function succeeds, access to the named resource with the name given by the argument *<resource>* is granted. Otherwise, **rump_enqueue** returns **-1** with *<errno>* set to the system error code.

If any other task currently has access to the resource, the calling task waits until the resource becomes free. This waiting is done in a first-in/first-out fashion to guarantee equal access to all tasks.

## ERRORS REPORTED

| | |
|---|---|
| ENOENT | The named resource does not exist |
| EINTR | The task caught a signal, and that caused this function to abnormally end |

## NOTES

A resource name is a NULL terminated string of no more than 16 characters.

## SEE ALSO

C Library: *rump_create(), rump_destroy(), rump_dequeue()*

# sbrk

Change the memory allocation of the data segment.

## SYNOPSIS

```
#include <errno.h>
char *sbrk(incr)
     int       incr;
```

### Arguments

| | |
|---|---|
| <incr> | The number of bytes to enlarge or shrink the data segment |

### Returns

If successful, the end-of-segment address of the data segment before it was enlarged or shrunk, otherwise *(char *)* **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **sbrk** function enlarges or shrinks the memory allocation of the data segment of the current task by *<incr>* bytes. If *<incr>* is positive, it enlarges the segment. If *<incr>* is negative, it shrinks the segment. If *<incr>* is zero, it does not change memory allocation of the segment. If **sbrk** succeeds, it returns the end-of-segment address for the data segment before the function changed the memory allocation of the segment. If *<incr>* is positive, this is the address of the first byte of newly allocated memory. If *<incr>* is negative, this address is meaningless, since it references memory that is out of the address space of the task. If *<incr>* is zero, this address is the current end-of-segment address of the data segment. Otherwise, **sbrk** returns *(char *)* **-1** with *<errno>* set to the system error code indicating the error.

The **sbrk** function fails if it could not allocate enough memory to make the data segment larger by *<incr>* bytes. **Sbrk** also fails if *<incr>* is negative and the absolute value of *<incr>* is larger than the number of bytes allocated to the data segment.

## ERRORS REPORTED

| | |
|---|---|
| ENOMEM | The function can not allocate enough memory to enlarge the data segment by <*incr*> bytes, or there is not enough memory allocated to the segment to shrink it by the requested number of bytes |

## NOTES

The **sbrk** function does not change the memory allocation of the data segment if it reports an error.

The end-of-segment address is the lowest address that is higher than the highest address of memory allocated to the segment.

The function returns the current end-of-segment address without changing the segment's memory allocation if <*incr*> is zero.

## SEE ALSO

C Library: *calloc(), EDATA, free(), malloc(), realloc()*

System Call: *brk(), cdata()*

# scanf

Read and interpret formatted data from stdin.

## SYNOPSIS

```
#include <stdio.h>
int scanf(format [, addrlist])
     char     *format;
```

### Arguments

| | |
|---|---|
| <format> | The address of a character-string containing a format description |

### Returns

The number of items in the address-list *<addrlist>* that it successfully assigns or EOF if an error occurs before it assigns any data

## DESCRIPTION

The **scanf** function reads and interprets data from the standard I/O stream *<stdin>*, according to the format description in the character-string referenced by *<format>*. Following the argument *<format>* in the argument list, **scanf** expects a list of address of variables to receive the values it generates from the data it reads from *<stdin>*, if any. The function returns as its result the number of assignments is makes, or EOF if it encounters an error before making the first assignment.

The *<format>* argument is a character-string containing a format description, which describes the format of the data read from *<stdin>*. The format description consists of literal characters, white-space characters, and field descriptions, in any sequence.

Literal characters are all characters that are not white-space characters (as defined by **isspace()**), and not part of field descriptions. A literal character tells **scanf** to match that character with the next character read from *<stdin>*. If it does not match exactly, **scanf** ends.

White-space characters are the space (′ ′), end-of-line (′\n′), horizontal-tab (′\t′), form-feed (′\f′), and carriage-return (′\r) characters. A white-space character tells **scanf** to read and consume characters from *<stdin>* until it reaches a character which is not a white-space character or it reaches the end of the data. The next character available from *<stdin>* is the next character which is not a white-space character. The **scanf** function does nothing with a white-space character in the format description if the next character from *<stdin>* is not a white-space character.

A field description tells **scanf** how to interpret the next character or characters read from *<stdin>*. The field description tells **scanf** the maximum number of characters to read, the form of the characters read, the type of value to assign any result to, and whether to perform an assignment. A field description has this syntax:

```
    %[*][<width>][<flags>]<type>
```

The *'%'* character introduces the field description. The *'*'* character tells **scanf** to suppress assigning the interpreted value to a variable. The *<width>* part tells **scanf** the maximum number of characters to read to satisfy the field (including leading white-space characters, if the field type skips leading white-space characters). The *<flags>* part alters the type of assignment made by **scanf**, and may be the 'h' or the 'l' character. The *<type>* part defines the type of the field and may be any one of the characters in the string: **cdefgosux%[**.

For a complete description of the field types of **scanf**, see the manual pages for the standard I/O function **fscanf()**.

## NOTES

The most common mistake made when using **scanf** is passing to the function the values of the variables to receive the results of this function, instead of the addresses of those variables.

The include-file *<stdio.h>* defines this function, other functions, macros, and constants used by standard I/O.

The C library contains two versions of the **fscanf()** function, which is used to implement this function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the function **pffinit()**. Otherwise, it loads the version which contains no floating-point conversions.

## SEE ALSO

C Library: *fdopen(), fopen(), fprintf(), fscanf(), pffinit(), printf(), sprintf(), sscanf(), stdin*

Commands: **cc**

# seekdir

Change the current position of a directory-stream.

## SYNOPSIS

```
#include <sys/dir.h>
void seekdir(pdir, pos)
     DIR      *pdir;
     long      pos;
```

### Arguments

| | |
|---|---|
| <pdir> | A reference to a directory-stream |
| <pos> | The new position for the directory-stream |

### Returns

Void

## DESCRIPTION

The **seekdir** function changes the current position of the directory-stream referenced by *<pdir>* to the position *<pos>. The next read-operation requested on the directory-stream <pdir> reads the directory entry at the offset <pos> from the beginning of the directory.*

## NOTES

The include-file *<sys/dir.h>* contains definitions for the data types, structures, constants, and functions needed to read directories.

The **seekdir** function does not perform any validity checks on the specified position.

## SEE ALSO

C Library: *closedir(), opendir(), readdir(), rewinddir(), telldir()*

## set_ftm

Change the last-modification time of a file.

## SYNOPSIS

```
#include <errno.h>
int set_ftm(pathnam, ptime)
     char      *pathnam;
     long      *ptime;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname for the file whose modification time is to change |
| <ptime> | The address of the value to set as the modification time for the file |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **set_ftm** function changes the last-modification time for the file reached by the pathname in the character-string referenced by *<pathnam>* to the system-time value referenced by *<ptime>*. The function requires that the current effective user be the system manager. The **set_ftm** function returns zero if it successfully changes the modification time of the file, otherwise **set_ftm** returns **-1** with *<errno>* set to the system error code.

The **set_ftm** function fails if the path in *<pathnam>* can not be followed or contains a file that is not a directory. It also fails if the pathname does not exist, the file reached by the pathname is currently open, or the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EBUSY | The specified file is not currently open |
| ENOENT | The pathname does not reach a file |
| EINVAL | An argument to the function is invalid |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The **set_ftm** function is obsolete and is included only for compatibility with older versions of the C Language. New applications should use **utime()**.

The operating system measures time as a count of seconds since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

The **set_ftm** function does not compare the new modification time with the creation time or the current time, so it is possible to set the modification time of the file to before its creation date or to some time in the future.

## SEE ALSO

System Call: *fstat(), stat(), utime()*

Commands: **dir**

## set_high_address_mask

Set the hardware high address mask register

## SYNOPSIS

```
#include <errno.h>
int set_high_address_mask(mask_value)
```

### Arguments

int mask_value

### Returns

If successful zero, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **set_high_address_mask** function sets the hardware high address mask register for the task. Each task has its own value used for this register, which defaults to 0xFFFFFFFF. This value corresponds to 32 significant address bits. If a task needs fewer than 32 bits, this function may be used to set a mask that indicates the number of significant bits. For example, the value 0x00FFFFFF indicates only 24 address bits are significant.

## ERRORS REPORTED

None

## NOTES

Setting this register may cause some invalid addresses to be treated as valid since some upper bits are being ignored.

Only bits 24-31 are significant in the mask value. Bits 0-23 are forced to be 0x00FFFFFF by the system.

## SEE ALSO

None

# setbuf

Set buffering attributes of a stream.

## SYNOPSIS

```
#include <stdio.h>
void setbuf(stream, buf)
     FILE     *stream;
     char     *buf;
```

### Arguments

| | |
|---|---|
| <stream> | The standard I/O stream whose buffer characteristics are being set |
| <buf> | The address of the buffer to use as the stream's buffer, or *(char *) NULL* if the stream is to be unbuffered |

### Returns

Void

## DESCRIPTION

The **setbuf** function sets the buffering characteristics for the standard I/O stream referenced by *<stream>*. If *<buf>* is *(char *) NULL*, the stream is set for unbuffered I/O. Otherwise, the stream is set for buffered I/O with *<buf>* set as address of the buffer to use for the buffered I/O.

## NOTES

The buffer whose address is *<buf>* is assumed to contain at least BUFSIZ bytes. The include-file *<stdio.h>* contains this and other definitions for standard I/O.

The **setbuf** function should only be used before any I/O is performed on the stream. If I/O has been performed on the stream, the current buffering is lost.

## SEE ALSO

C Library: *fdopen(), fopen()*

## setjmp

Setup for a non-local goto.

## SYNOPSIS

```
#include <setjmp.h>
int setjmp(env)
     jmp_buf   env;
```

### Arguments

| | |
|---|---|
| <env> | The value to receive the current environmental information |

### Returns

Zero when returning from **setjmp()**, non-zero when the result of a **longjmp()**

## DESCRIPTION

The **setjmp** function saves the current environmental information in the argument *<env>* so that a subsequent call to **longjmp()** with *<env>* as its argument results with execution continuing as though the **setjmp()** call had returned. The effect of a **longjmp()** using *<env>* as its argument is that of a *goto* from the **longjmp()** call to the **setjmp()** call.

A 0 result indicates that **setjmp()** is returning after setting the argument *<env>* with the current environmental information. A non-zero result indicates that **longjmp()** was called with an argument *<env>*.

## NOTES

The scope calling the **setjmp** function must not have returned by the time **longjmp()** is called with the argument *<env>* or the result of the **longjmp()** call is unpredictable.

Values residing in registers (those defined as *register* variables that have had registers assigned to them) revert to their value at the time of the **setjmp()** call when **longjmp()** is called with the argument *<env>*.

The argument *<env>* is actually the address of a structure for the current environmental information. The include-file *<setjmp.h>* contains the *typedef* for *jmp_buf* and other information used by **setjmp()** and **longjmp()**.

## SEE ALSO

C Library: *longjmp()*

# setpwent

Reset password-file handling.

## SYNOPSIS

```
#include <pwd.h>
void setpwent();
```

### Arguments

None

### Returns

Void

## DESCRIPTION

The **setpwent** function resets password-file handling initiated by **getpwent()**, **getpwnam()**, or **getpwuid()**. The **setpwent** function reinitializes the resources allocated by and rewinds the files opened by those routines.

## NOTES

The **setpwent** function does nothing if **getpwent()**, **getpwnam()**, or **getpwuid()** has not been called or **endpwent()** has been called since the last call to one of those routines.

## SEE ALSO

C Library: *endpwent(), getpwent(), getpwnam(), getpwuid()*

## setuid

Change both the user-ID and the effective user-ID.

## SYNOPSIS

```
#include <errno.h>
int setuid(uid)
     int        uid;
```

### Arguments

| | |
|---|---|
| <uid> | The user-ID of the new user and effective user |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **setuid** function changes the current user-ID and the current effective user-ID of the task to *<uid>*. The **setuid** function expects either the current user or the current effective user to be the system manager. The function returns zero as its result if it successfully changes the user-ID and effective user-ID of the task. Otherwise, **setuid** returns **-1** with *<errno>* set to the system error code.

The **setuid** function fails if neither the current user nor the current effective user is the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | Neither the current user nor the current effective user is the system manager |

## SEE ALSO

System Call: *geteuid(), getuid()*

Commands: **login**

# signal

Change the signal-handling address for a specific signal in the current task.

## SYNOPSIS

```
#include <errno.h>
#include <sys/signal.h>
int (*signal(signum, handler))()
     int       signum;
     int      (*handler)();
```

### Arguments

| | |
|---|---|
| <signum> | The signal number for the signal handling being changed |
| <handler> | The new signal-handling address for the specified signal |

### Returns

The previous signal-handling address for the specified signal if successful, otherwise *(int (*)())* **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **signal** function changes the signal-handling address for the specified signal *<signum>* in the current task to the function with the address *<handler>*. If **signal** succeeds, it returns as its result the previous signal-handling address for the specified function. If **signal** fails, it returns as its result *(int (*)())* **-1** with *<errno>* set to the system error code.

The value SIG_IGN is a special signal-handling address which, if passed to this function as the signal-handling address *<handler>*, tells the **signal** function that the task is to ignore the specified signal. If **signal** returns this value, the task was ignoring the specified signal. The value SIG_DFL is a special signal-handling address which, if passed to **signal** as the signal-handling address *<handler>*, tells the function that the task is to take default action if it receives the specified signal. For all signals except SIGDUMP, this action is task termination. If **signal** returns this value, the task would have terminated if it received the specified signal. The include-file *<sys/signal.h>* contains the definitions for SIG_IGN and SIG_DFL.

That include-file also defines constants for each of the sixty-three signals defined by the operating system. These constants are:

| | | |
|---|---|---|
| SIGHUP | 1 | Hang-up |
| SIGINT | 2 | Keyboard |
| SIGQUIT | 3 | Quit |
| SIGEMT | 4 | A-line ($Axxx) emulation trap |
| SIGKILL | 5 | Task kill |
| SIGPIPE | 6 | Broken pipe |
| SIGSWAP | 7 | Swap error |
| SIGTRACE | 8 | Trace |
| SIGTIME | 9 | Time limit |
| SIGALRM | 10 | Alarm |
| SIGTERM | 11 | Task terminate |
| SIGTRAPV | 12 | TRAPV instruction |
| SIGCHK | 13 | CHK instruction |
| SIGEMT2 | 14 | F-line ($Fxxx) emulation trap |
| SIGTRAP1 | 15 | TRAP #1 instruction |
| SIGTRAP2 | 16 | TRAP #2 instruction |
| SIGTRAP3 | 17 | TRAP #3 instruction |
| SIGTRAP4 | 18 | TRAP #4 instruction |
| SIGTRAP5 | 19 | TRAP #5 instruction |
| SIGTRAP6 | 20 | TRAP #6-14 instruction |
| SIGPAR | 21 | Parity error |
| SIGILL | 22 | Illegal instruction |
| SIGDIV | 23 | Division by 0 |
| SIGPRIV | 24 | Privileged instruction |
| SIGADDR | 25 | Address error |
| SIGDEAD | 26 | A child task terminated |
| SIGWRIT | 27 | Write to read-only memory |
| SIGEXEC | 28 | Execute from stack or data space |
| SIGBND | 29 | Segmentation violation |
| SIGUSR1 | 30 | User-defined signal #1 |
| SIGUSR2 | 31 | User-defined signal #2 |
| SIGUSR3 | 32 | User-defined signal #3 |
| SIGABORT | 33 | Program abort |
| SIGSPLR | 34 | Spooler signal |
| SIGINPUT | 35 | Input is ready |
| SIGDUMP | 36 | Take memory dump |
| SIGVEN14 | 62 | Millisecond Alarm |
| SIGVEN15 | 63 | Mouse/keyboard event interrupt |

## ERRORS REPORTED

| | |
|---|---|
| EINVAL | The <*signum*> value is not a valid signal number |

## NOTES

The **signal()** function is a function returning a pointer to a function returning an *int.*

The <*handler*> argument is a pointer to a function returning an *int.*

The **signal** function does not verify the argument <*handler*> to ensure that no memory-violation or bus-error occurs if the specified signal is caught.

The signals SIGTIME and SIGINPUT are not currently implemented.

The operating system produces a core image in a file called *core* in the working directory if the default action (termination) is taken by a task on receipt of certain signals and other conditions are met. Those certain signals are: SIGABORT, SIGADDR, SIGBND, SIGCHK, SIGDUMP, SIGEMT, SIGEMT2, SIGDIV, SIGDUMP, SIGEXEC, SIGILL, SIGQUIT, SIGPAR, SIGPRIV, SIGTIME, SIGTRAP1, SIGTRAP2, SIGTRAP3, SIGTRAP4, SIGTRAP5, SIGTRAP6, SIGTRAPV, and SIGWRIT. The operating system generates a core image file only if there is an existing file in the working directory called *core* that gives the current effective user writing permission or the working directory gives the current effective user writing permission.

The default action for the SIGDUMP signal is to create a dump file and return control to the task. The task is not terminated.

The operating system does not permit any task to catch either a SIGABORT or a SIGKILL signal.

The operating system does not permit tasks to ignore some signals. Those signals are: SIGABORT, SIGADDR, SIGBND, SIGEXEC, SIGILL, SIGKILL, SIGPAR, SIGPRIV, SIGSWAP, SIGTIME, SIGWRIT.

The operating system sets the initial state of one signal to SIG_IGN. That signal is SIGDEAD.

The operating system does not reset either the SIGDEAD or the SIGTRACE interrupt when it occurs.

## SEE ALSO

System Call: *kill()*

Commands: **int**

# sin

Calculate the sine of an angle.

## SYNOPSIS

```
#include <math.h>
double sin(r)
     double    r;
```

### Arguments

| | |
|---|---|
| <r> | The angle whose sine is to be computed |

### Returns

The sine of the angle *<r>*

## DESCRIPTION

The **sin** function calculates the sine of the angle *<r>*. It returns that value as its result.

The **sin** function interprets the value *<r>* as an angle expressed in radians, and the function returns a result between -1.0 and 1.0 inclusive.

## SEE ALSO

C Library: *asin(), cos(), tan()*

## sinh

Calculate the hyperbolic sine of a value.

## SYNOPSIS

```
#include <math.h>
double sinh(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value whose hyperbolic sine is to be computed |

### Returns

The hyperbolic sine of the argument *<x>*

## DESCRIPTION

The **sinh** function calculates the hyperbolic sine of the value *x*. The hyperbolic sine of *<x>* is defined as (exp(x) - exp(-x))/2. It returns that value as its result.

The **sinh** function detects a range error if the magnitude of the hyperbolic sine of *<x>* is larger than can be represented by the data type *double*. If **sinh** detects a range error, it calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. It sets the *<type>* element of the structure to OVERFLOW, *<name>* to the address of the character-string *<sinh>*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, **sinh** sets *<errno>* to ERANGE. The return value, which is system-dependent, is given in the manual page for **kill()**. If **matherr()** returns something other than 0, the **sinh** function returns the value *retval* found in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *exp(), matherr()*

## sleep

Suspend execution for an interval.

## SYNOPSIS

```
unsigned int sleep(time)
     unsigned int   time;
```

### Arguments

| | |
|---|---|
| <time> | The maximum number of seconds to suspend execution |

### Returns

The number of seconds remaining in the requested interval

## DESCRIPTION

The **sleep** function requests that the execution of the current task be suspended for the number of seconds specified by the *<time>* argument. The **sleep** function returns after the requested interval passes or an alarm-, hangup-, keyboard-, or quit-interrupt is caught. It returns as its result the number of seconds remaining in the requested sleep interval.

The **sleep** function is implemented using the **alarm** system call. **Sleep** requests that an alarm-interrupt be sent to the current task in *<time>* seconds, then pauses using the **pause()** function, waiting for a signal. The function knows about an alarm-interrupt request armed before the function is called. If the armed alarm-interrupt request is scheduled to take place during the sleep interval, the function pauses for time remaining on the armed alarm-interrupt request. Then, if that interval passes completely, it resignals the alarm-interrupt so the user can handle it. Otherwise (or if the armed alarm-interrupt request is scheduled to take place after the sleep interval is complete), upon return from the **pause()** function, **sleep** rearms the alarm-interrupt for the time remaining. **Sleep** then restores the signaling information for the alarm-interrupt to the state before **sleep** was called.

## NOTES

Requesting a sleep interval <*time*> of 0 results in the task pausing until the next signal.

## SEE ALSO

System Call: *alarm(), signal(), wait()*

Commands: **sleep**

# sprintf

Generate a character-string containing formatted data.

## SYNOPSIS

```
#include <stdio.h>
int sprintf(string, format [,arglist])
     char     *string;
     char     *format;
```

### Arguments

| | |
|---|---|
| <string> | The address of a buffer to contain the generated string |
| <format> | The address of a character-string containing a format description |

### Returns

The number of characters written to *<string>* or EOF if an error occurred

## DESCRIPTION

The **sprintf** function generates characters from the format description in the character-string referenced by *<format>* and the arguments in the argument-list *<arglist>*, if any, writes these characters into the buffer with the address *<string>*, then appends a null-character onto those generated characters in that buffer. The **sprintf** function returns as its result the length of the generated character-string.

The format description in the character-string referenced by *<format>* contains literal characters and field descriptions. The **sprintf** function copies literal characters to character-string with no interpretation. The function interprets field descriptions to determine what characters it generates, what type of argument it consumes, if any, from the argument list *<arglist>*, and the type of conversion it performs. The number of arguments and the type of the arguments in the argument list *<arglist>* depends on the format description. The argument list can be omitted.

For a complete description of the *<format>* argument, see the manual page for **fprintf()**.

## NOTES

The **sprintf** function assumes that the buffer whose address is *<string>* is large enough to hold the character-string it generates.

The include-file *<stdio.h>* defines this function and other functions and constants available in standard I/O. This file must be included in the C source before the first reference to this function.

The C library contains two versions of this function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the **pffinit()** function. Otherwise, **sprintf** loads the version that contains no floating-point conversions.

## SEE ALSO

C Library: *ecvt(), fcvt(), fdopen(), fopen(), fprintf(), fputc(), fscanf(), gcvt(), pffinit(), printf(), scanf(), sprintf(), sscanf(), stdout*

Commands: **cc**

# sqrt

Calculate the square root of a value.

## SYNOPSIS

```
#include <math.h>
double sqrt(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The value whose square root is to be computed |

### Returns

The square root of the argument *<x>*

## DESCRIPTION

The **sqrt** function calculates the square root of the value *<x>*. It returns the calculated value as its result.

The **sqrt** function demands that the argument *<x>* be greater than or equal to zero. If *<x>* is less than zero, **sqrt** detects a domain error and calls **matherr()**, passing to it the address of a filled *<struct>* exception structure. **Sqrt** sets the *<type>* element of the structure to DOMAIN, *<name>* to the address of the character-string *<sqrt>*, and *<arg1>* to *<x>*.

If **matherr()** returns 0, **sqrt** writes the message

```
        sqrt() error:  Negative argument
```

to the standard error I/O stream *<stderr>* and sets *<errno>* to EDOM. The return value, which is system-dependent, is given in the manual page for **kill()**. If **matherr()** returns something other than 0, the **sqrt** function returns the value *retval* in the *<struct>* exception structure as its result.

## SEE ALSO

C Library: *matherr()*

# srand

Set the seed of the random number generator.

## SYNOPSIS

```
void srand(seed)
     int       seed;
```

### Arguments

| | |
|---|---|
| <seed> | The seed for the random number generator |

### Returns

Void

## DESCRIPTION

The **srand** function sets the seed of the pseudo-random number generator to a value generated from the argument *<seed>*. The value generated is that of the low 15 bits of the argument.

## NOTES

The seed is the value used to generate the next random number.

The random number generating function **rand()** always generates the same sequence of random numbers from a particular seed.

## SEE ALSO

C Library: *rand(), rrand()*

## sscanf

Interpret formatted data from a character-string.

## SYNOPSIS

```
#include <stdio.h>
int sscanf(string, format [, addrlist])
     char      *string;
     char      *format;
```

### Arguments

| | |
|---|---|
| <string> | The string containing data to interpret |
| <format> | The address of a character-string containing a format description |

### Returns

The number of items in the address-list *<addrlist>* that it successfully assigns or EOF if an error occurs before it assigns any data

## DESCRIPTION

The **sscanf** function interprets data from the character-string referenced by the argument *<string>*, according to the format description in the character-string referenced by *<format>*. Following the argument *<format>* in the argument list, **sscanf** expects a list of addresses of variables to receive the values it generates from the characters in the data character-string, if any. The **sscanf** function returns as its result the number of assignments is makes, or EOF if it encounters an error before making the first assignment.

The argument *<format>* is a character-string containing a format description, which describes the format of the characters in the data character-string. The format description consists of literal characters, white-space characters, and field descriptions, in any sequence.

Literal characters are all characters that are not white-space characters (as defined by **isspace()**), and not part of field descriptions. A literal character tells the function to match that character with the next character in the data character-string. If it does not match exactly, the function ends.

White-space characters are the space (' '), end-of-line ('\n'), horizontal-tab ('\t'), form-feed ('\f'), and carriage-return ('\r) characters. A white-space character tells **sscanf** to skip characters in the data character-string until it reaches a character that is not a white-space character or it reaches the end of the data. The next character available from the data character-string is the next character that is not a white-space character. The **sscanf** function does nothing with a white-space character in the format description if the next character from the data character-string not a white-space character.

A field description tells **sscanf** how to interpret the next character or characters from the data character-string. It tells the function the maximum number of characters to get, the form of those characters, the type of value to assign any result to, and whether to perform an assignment. A field description has this syntax:

```
%[*][<width>][<flags>]<type>
```

The *'%'* character introduces the field description. The *'*'* character tells **sscanf** to suppress assigning the interpreted value to a variable. The *<width>* part tells **sscanf** the maximum number of characters to get to satisfy the field (including leading white-space characters, if the field type skips leading white-space characters). The *<flags>* part alters the type of assignment made by the function, and may be either the *'h'* or the *'l'* character. The *<type>* part defines the type of the field and may be any one of the characters in this string: **cdefgosux%[**.

For a complete description of the field types of the **sscanf** function, see the manual pages for the standard I/O function **fscanf()**.

## NOTES

The most common mistake made when using **sscanf** is passing to the function the values of the variables to receive the results of **sscanf**, instead of the addresses of those variables.

The include-file *<stdio.h>* defines **sscanf**, other functions, macros, and constants used by standard I/O.

The C library contains two versions of the **sscanf** function: one that contains floating-point conversions and one that contains no floating-point conversions. The **cc** command loads the version containing floating-point conversions only if the C source contains references to the one of the floating-point data types or a call to the **pffinit()** function. Otherwise, **cc** loads the version that contains no floating-point conversions.

## SEE ALSO

C Library: *fdopen(), fopen(), fprintf(), fscanf(), pffinit(), printf(), scanf(), sprintf(), stdin*

Commands: **cc**

# stack

Check and expand memory allocated to the stack segment of the task.

## SYNOPSIS

```
#include <errno.h>
int stack(nbytes)
    int      nbytes;
```

### Arguments

| | |
|---|---|
| <nbytes> | The number of bytes that the stack is expected to grow |

### Returns

Zero if the stack segment has enough space to contain a program stack enlarged by *<nbytes>* bytes, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **stack** function guarantees that the the task has enough memory allocated to its stack segment so that the program stack can expand by *<nbytes>* bytes. If the segment is not large enough, **stack** allocates enough memory to the segment so that the stack can expand by the specified amount.

The **stack** function returns zero if the stack segment has enough space allocated to it to contain a program stack enlarged by *<nbytes>* bytes. Otherwise, **stack** returns **-1** with *<errno>* set to the system error code.

The **stack** function fails if can not allocate more memory to the stack segment of the task.

## ERRORS REPORTED

| | |
|---|---|
| ESTOF | The stack segment of the task is as large as it can get |
| EVFORK | The task shares its memory with its pary an may not call this function. |

## NOTES

The C-compiler automatically generates code that ensures stack integrity.

## SEE ALSO

Commands: cc

## stat

Get the status of a file.

## SYNOPSIS

```
#include <errno.h>
#include <sys/stat.h>
int stat(pathnam, bufad)
     char          *pathnam;
     struct stat   *bufad;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname for the file to examine |
| <bufad> | The address of the structure to contain the file´s status |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **stat** function examines the file reached by the pathname in the character-string referenced by *<pathnam>* and writes information describing the status of that file into the structure whose address is *<bufad>*. The **stat** function returns zero as its result if it successfully gets the status of the file. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **stat** function fails if the path in *<pathnam>* cannot be followed or if it contains a file that is not a directory. **Stat** also fails if the pathname does not reach a file.

The following structure is defined in the include-file *<sys/stat.h>* and defines the format of the data describing the status of the file:

```
struct stat
{
     short     st_dev;
     short     st_ino;
     char      st_filler;
     char      st_mode;
     char      st_perm;
     char      st_nlink;
     short     st_uid;
     long      st_size;
     long      st_mtime;
     long      st_spr;
};
```

The *st_dev* value is the device number of the device containing the file. *st_ino* is the file descriptor number (FDN) on the device describing the file. *st_filler* is an unused byte. *st_mode* is a bit-string describing the type of the file, (described below). *st_perm* is a bit-string describing the permissions of the file, (described below). *st_nlink* is the number of links to the file (since the maximum number that may be stored in a character field is 127, the maximu link count is 127). *st_uid* is the owner-ID of the file. *st_size* is the size of the file, in bytes. *st_mtime* is the last modification date and time for the file, in system-time. *st_spr* is unused.

The following constants, defined in the include-files *<sys/stat.h>* and *<sys/modes.h>*, define the data in the bit-string *st_mode* (which describes the type of file):

```
S_IFMT    0x4F
S_IFREG   0x01
S_IFBLK   0x03
S_IFCHR   0x05
S_IFPTY   0x07
S_IFDIR   0x09
S_IFPIPE  0x41
```

The constant S_IFMT is a mask that, when anded with the value *st_mode*, yields the file type. After anding with the constant S_IFMT, *st_mode* produces S_IFREG if the file is a regular file, S_IFBLK if the file is a block-special file (block device), S_IFPTY if the file is a pseudo tty device, S_IFCHR if the file is a character-special file (character device), S_IFDIR if the file is a directory, or S_IFPIPE if the file is a pipe.

The following constants, also defined in the include-files *<sys/stat.h>* and *<sys/modes.h>*, define the data in the bit-string *st_perm* (which describes the access permissions of the file):

```
S_IREAD   0x01
S_IWRITE  0x02
S_IEXEC   0x04
S_IOREAD  0x08
S_IOWRITE 0x10
S_IOEXEC  0x20
S_ISUID   0x40
```

The value S_IREAD grants reading permission to the owner of the file. S_IWRITE grants writing permission to the owner. S_IEXEC grants searching permission to the owner if the file is a directory, otherwise S_IEXEC grants execution permission. The value S_IOREAD grants reading permission to users other than the owner of the file. S_IOWRITE grants writing permission to others. S_IOEXEC grants searching permission to others if the file is a directory, otherwise S_IOEXEC grants execution permission. The value S_ISUID causes the effective user-ID to be changed to that of the owner of the file when the program contained in the file is executed.

## ERRORS REPORTED

| | |
|---|---|
| EMSDR | Could not follow the path to the file |
| ENOENT | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The *<sys/modes.h>* include-file does not need to be included if the *<sys/stat.h>* include-file is included, since *<sys/stat.h>* contains *<sys/modes.h>*.

## SEE ALSO

System Call: *creat(), dup(), dup2(), fstat(), link(), open(), pipe(), utime()*

Commands: **dir**

# stderr

Standard error stream for standard I/O.

## SYNOPSIS

```
#include <stdio.h>
FILE *stderr;
```

## DESCRIPTION

The **stderr** value references the standard error stream for standard I/O. The **stderr** value may be used anywhere a FILE * is required.

The **stderr** value must not be defined explicitly, since it is defined by the *<stdio.h>* include-file. **Stderr** may not be modified. Unless explicitly changed by the **setbuf()** function, this stream is an unbuffered output stream. The stream may be detached from the original standard error file and attached to another file by using the **freopen()** function.

## NOTES

Closing this stream, along with *<stdin>* and *<stdout>*, cause the current program to detach from its controlling terminal.

## SEE ALSO

C Library: *fclose(), freopen(), perror(), stdin, stdout*

# stdin

Standard input stream for standard I/O.

## SYNOPSIS

```
#include <stdio.h>
FILE *stdin;
```

## DESCRIPTION

The **stdin** value references the standard input stream for standard I/O. The **stdin** value may be used anywhere a FILE * is required.

The **stdin** value must not be defined explicitly, since it is defined by the include-file *<stdio.h>*. **Stdin** may not be modified. Unless explicitly changed by the **setbuf()** function, this stream is buffered. The stream may be detached from the original standard input file and attached to another file by using the **freopen()** function.

## NOTES

Closing this stream, along with *<stderr>* and *<stdout>*, cause the current program to detach from its controlling terminal.

## SEE ALSO

C Library: *fclose(), freopen(), getchar(), gets(), scanf(), stderr, stdout*

# stdout

Standard output stream for standard I/O.

## SYNOPSIS

```
#include <stdio.h>
FILE *stdout;
```

## DESCRIPTION

The **stdout** value references the standard output stream for standard I/O. **Stdout** may be used anywhere a FILE * is required.

The **stdout** value must not be defined explicitly, since it is defined by the include-file *<stdio.h>*. The **stdout** value may not be modified. Unless explicitly changed by the **setbuf()** function, this stream is buffered. The stream may be detached from the original standard output file and attached to another file by using the **freopen()** function.

## NOTES

Closing this stream, along with *<stderr>* and *<stdin>*, causes the current program to detach from its controlling terminal.

## SEE ALSO

C Library: *fclose(), freopen(), printf(), putchar(), puts(), stderr, stdin*

## stime

Set the system-time value.

## SYNOPSIS

```
#include <errno.h>
int stime(ptime)
     long     *ptime;
```

### Arguments

| | |
|---|---|
| <ptime> | The address of the value to set as the new system-time value |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **stime** function changes the system-time value of the operating system (the current time-of-day) to the value referenced by *<ptime>*. The **stime** function requires that the current effective user be the system manager. The function returns zero as its result if it successfully sets the system-time value. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **stime** function fails if the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |

## NOTES

The operating system represents the time of day as the number of seconds that has elapsed since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

## SEE ALSO

System Call: *time(), times()*

Command: **date**

# _stol2

Convert *short* integers to two-byte integers.

## SYNOPSIS

```
void _stol2(cp, sp, n)
     char     *cp;
     short    *sp;
     int       n;
```

### Arguments

| | |
|---|---|
| <cp> | The address of the buffer to contain the two-byte integers |
| <sp> | The address of the buffer containing the *short* integers |
| <n> | The number of values to convert |

### Returns

Void

## DESCRIPTION

The _stol2 function converts *<n>* *short* integers in the array referenced by *<sp>* to two-byte integers, saving the converted values packed into the array of *<char>* referenced by *<cp>*. The _stol2 function returns no result.

## NOTES

The _stol2 function is typically used to avoid addressing problems resulting from misaligned addresses.

## SEE ALSO

C Library: *_l2tos(), l3tol(), _l4tol(), ltol3(), _ltol4()*

## strcat

Concatenate one character-string onto another.

## SYNOPSIS

```
#include <string.h>
char *strcat(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the target character-string |
| <s2> | The address of the character-string to concatenate onto *<s1>* |

### Returns

*<s1>*

## DESCRIPTION

The **strcat** function appends a copy of the character-string referenced by *<s2>* onto the character-string referenced by *<s1>*. The **strcat** function returns *<s1>* as its result.

## NOTES

The resulting character-string is always terminated with a null-character.

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strncat()*

## strchr

Find the first occurrence of a character in a character-string.

## SYNOPSIS

```
#include <string.h>
char *strchr(s, c)
     char     *s;
     char      c;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to search |
| <c> | The character to search for |

### Returns

The address of the first occurrence of the character in the string, or *(char *)* NULL if the string does not contain the character

## DESCRIPTION

The **strchr** function searches the character-string with the address *<s>* for the first occurrence of the character *<c>*. If the string contains the character, **strchr** returns as its result the address of the first occurrence of the character. Otherwise, **strchr** returns *(char *)* NULL.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *index(), strrchr()*

# strcmp

Compare two character-strings.

## SYNOPSIS

```
#include <string.h>
int strcmp(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the first string to compare |
| <s2> | The address of the second string to compare |

### Returns

A value less than, equal to, or greater than zero, if the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the character-string referenced by *<s2>*

## DESCRIPTION

The **strcmp** function lexicographically compares the character-string referenced by *<s1>* with the character-string referenced by *<s2>* and returns as its result a value that indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the character-string referenced by *<s2>*.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strncmp()*

# strcmpci

Compare two character-strings (case insensitive).

## SYNOPSIS

```
#include <string.h>
int strcmpci(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the first string to compare |
| <s2> | The address of the second string to compare |

### Returns

A value less than, equal to, or greater than zero, if the character-string referenced by <*s1*> is lexicographically less than, equal to, or greater than the character-string referenced by <*s2*>

## DESCRIPTION

This function lexicographically compares the character-string referenced by <*s1*> with the character-string referenced by <*s2*> using case insensitive comparisons and returns as its result a value which indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the character-string referenced by <*s1*> is lexicographically less than, equal to, or greater than the character-string referenced by <*s2*>.

## NOTES

The include-file <*string.h*> defines the string-handling functions in the C library.

This is not a standard System V library function.

## SEE ALSO

C Library: *strcmp(), strncmp(), strncmpci()*

# strcpy

Copy a character-string.

## SYNOPSIS

```
#include <string.h>
char *strcpy(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the target buffer |
| <s2> | The address of the character-string to copy |

### Returns

*<s1>*

## DESCRIPTION

The **strcpy** function copies the character-string referenced by *<s2>* into the buffer with the address *<s1>*. The **strcpy** function returns the address of the target buffer as its result.

## NOTES

The result of the **strcpy** function is always a null-terminated character-string.

The standard C library does not define the behavior of overlapping data movement, so using overlapping data movement may result in differing behavior on different systems.

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strncpy()*

# strcspn

Determine the unlike character-count.

## SYNOPSIS

```
#include <string.h>
int strcspn(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the character-string to examine |
| <s2> | The address of the character-string containing the characters to search for |

### Returns

The length of the initial segment of *<s1>* containing none of the characters found in *<s2>*

## DESCRIPTION

The **strcspn** function examines the character-string with the address *<s1>* and determines the length of the initial character segment containing none of the characters found in the character-string with the address *<s2>*. The **strcspn** function returns this count as its result.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strspn()*

# strerror

Return a pointer to a message describing the specified error number.

## SYNOPSIS

```
char *strerror(e_num)
     int       e_num;
```

### Arguments

<e_num>    The error number whose associated error message should be returned.

A pointer to the error message associated with <e_num>.

## DESCRIPTION

If *<e_num>* is greater than or equal to zero but less than 512, the function gets the error message from the file **/gen/errors/system.**

If **<e_num>** is greater than or equal to 512 but less than 1024, it gets the message from the file **/gen/errors/local** using (*<e_num>* % 512) as the error number. If *<e_num>* is greater than or equal to 1024, the function gets the error message from a file whose name is of the form **/gen/errors/errorfile%4.4u** where "%4.4u" is replaced by ((*<e_num>*-1024) / 256), using (*<e_num>* % 256) as the error number.

## NOTES

This function returns a pointer to the message
`No message for error number =`
followed by the value of *<e_num>* if it could not find a message for the current value of *<e_num>*.

This function initializes the global variable **sys_nerr** and the global table **sys_errlist.**

This is an ANSI standard library function.

## SEE ALSO

C Library: *errno, _ierrmsg(), perror(), sys_errlist, sys_nerr*

## strlen

Determine the length of a character-string.

## SYNOPSIS

```
#include <string.h>
int strlen(s)
     char     *s;
```

### Arguments

| | |
|---|---|
| <s> | The address of a character-string |

### Returns

The number of characters in the character-string

## DESCRIPTION

The **strlen** function determines the length of character-string referenced by *<s>* and returns that value as its result.

The **strlen** function determines the length of the character-string by counting the number of characters that are not null-characters, beginning at the address *<s>*, and continuing until a null-character is found.

## NOTES

A null-string *("")* has a length of zero.

The include-file *<string.h>* defines the string-handling functions in the C library.

# strncat

Concatenate one character-string onto another.

## SYNOPSIS

```
    #include <string.h>
                 int       n;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the target character-string |
| <s2> | The address of the character-string to concatenate onto <s1> |
| <n> | The maximum number of characters to concatenate |

### Returns

<s1>

## DESCRIPTION

The **strncat** function appends at most <*n*> characters from the character-string referenced by <*s2*> onto the character-string referenced by <*s1*>. It returns as its result <*s1*>.

## NOTES

The function always appends a null-character onto the characters appended onto <*s1*> from <*s2*>. The include-file <*string.h*> defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strcat()*

## strncmp

Compare two character-strings.

## SYNOPSIS

```
#include <string.h>
int strncmp(s1, s2, n)
     char     *s1;
     char     *s2;
     int       n;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the first string to compare |
| <s2> | The address of the second string to compare |
| <n> | The maximum number of characters to compare |

### Returns

A value less than, equal to, or greater than zero, if the first *<n>* characters in the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the first *<n>* characters in the character-string referenced by *<s2>*

## DESCRIPTION

The **strncmp** function lexicographically compares a maximum of *<n>* characters from the character-string referenced by *<s1>* with a maximum of *<n>* characters of the character-string referenced by *<s2>*. The *strncmp* function returns as its result a value that indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the first *<n>* characters of the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the fist *<n>* characters of the character-string referenced by *<s2>*.

## NOTES

The include-file <*string.h*> defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strcmp()*

## strncmpci

Compare two character-strings (case insensitive).

## SYNOPSIS

```
#include <string.h>
int strncmpci(s1, s2, n)
     char      *s1;
     char      *s2;
     int        n;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the first string to compare |
| <s1> | The address of the second string to compare |
| <n> | The maximum number of characters to compare |

### Returns

A value less than, equal to, or greater than zero, if the first *<n>* characters in the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the first *<n>* characters in the character-string referenced by *<s2>*

## DESCRIPTION

This function lexicographically compares a maximum of *<n>* characters from the character-string referenced by *<s1>* with a maximum of *<n>* characters of the character-string referenced by *<s2>* and returns as its result a value which indicates the result of that comparison. The comparison is done ignoring the case of all letters. That value is less than, equal to, or greater than zero, indicating that the first *<n>* characters of the character-string referenced by *<s1>* is lexicographically less than, equal to, or greater than the fist *<n>* characters of the character-string referenced by *<s2>*.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

This is not a standard System V library function.

## SEE ALSO

C Library: *strcmp(), strcmpci(), strncmp()*

# strncpy

Copy a character-string.

## SYNOPSIS

```
    #include <string.h>
                int        n;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the target buffer |
| <s2> | The address of the character-string to copy |
| <n> | The maximum number of characters to copy |

### Returns

<s1>

## DESCRIPTION

The **strncpy** function copies characters from the character-string referenced by *<s2>* into the buffer whose address is *<s1>* until *<n>* characters have been copied. If a null is found in *<s2>* before *<n>* has been copied, then it will place nulls at the destination. **Strncpy** returns the address of the target buffer as its result.

## NOTES

The **strncpy** function does not append a null-character to the copied characters. The standard C library does not define the behavior of overlapping data movement, so using overlapping data movement may result in differing behavior on different systems. The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strcpy()*

# strpbrk

Find the first occurrence of any of a list of characters in a character-string.

## SYNOPSIS

```
#include <string.h>
char *strpbrk(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

|  |  |
|---|---|
| <s1> | The address of the character-string to search |
| <s2> | The address of the character-string containing the list of characters to search for |

### Returns

The address of the first occurrence of any of the characters in *<s2>* found in *<s1>*, or *(char *)* NULL if none of the characters in *<s2>* were found in *<s2>*

## DESCRIPTION

The **strpbrk** function searches the character-string with the address *<s1>* for the first occurrence of any character in the character-string with the address *<s2>* and returns as its result the address of that character in *<s1>*. If the function fails to find any of the characters in *<s2>* in the character-string *<s1>*, it returns *(char *)* NULL as its result.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strchr()*

# strrchr

Find the last occurrence of a character in a character-string.

## SYNOPSIS

```
#include <string.h>
char *strrchr(s, c)
     char     *s;
     char      c;
```

### Arguments

| | |
|---|---|
| <s> | The address of the character-string to search |
| <c> | The character to search for |

### Returns

The address of the last occurrence of the character in the string, or *(char *)* NULL if the string does not contain the character

## DESCRIPTION

The **strrchr** function searches the character-string with the address *<s>* for the last occurrence of the character *<c>*. If the string contains the character, the **strrchr** function returns as its result the address of the last occurrence of the character. Otherwise, the function returns *(char *)* NULL.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *rindex(), strchr()*

## strspn

Determine the like character-count.

## SYNOPSIS

```
#include <string.h>
int strspn(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the character-string to examine |
| <s2> | The address of the character-string containing the characters to search for |

### Returns

The length of the initial segment of *<s1>* containing only characters found in *<s2>*

## DESCRIPTION

The **strspn** function examines the character-string with the address *<s1>* and determines the length of the initial character segment containing only characters found in the character-string with the address *<s2>*. The **strspn** function returns this count as its result.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strcspn()*

# strstr

Find a substring within a character-string.

## SYNOPSIS

```
#include <string.h>
char *strstr(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the string in which to search |
| <s2> | The address of the substring to search for |

### Returns

The address of the substring *<s2>* in *<s1>* if found, or (char *) NULL if none

## DESCRIPTION

This function searches for the substring *<s2>* in the string *<s1>*. It returns as its result the address of the substring if found, or (char *) NULL if not.

## NOTES

The include-file *<string.h>* defines the string-handling functions in the C library.

This is an ANSI standard library function.

## SEE ALSO

C Library: *strspn(), strstrci(), strtok()*

## strstrci

Find a substring within a character-string (case insensitive).

### SYNOPSIS

```
#include <string.h>
char *strstrci(s1, s2)
   char   *s1;
   char   *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the string in which to search |
| <s2\|> | The address of the substring to search for |

The address of the substring *<s2>* in *<s1>* if found, or **(char *)** NULL if none

### DESCRIPTION

This function searches for the substring *<s2>* in the string *<s1>* using case insensitive compares. Upper case letters will match lower case letters. It returns as its result the address of the substring if found, or **(char *)** NULL if not.

### NOTES

The include-file **<string.h>** defines the string-handling functions in the C library.

This is not a standard System V library function.

### SEE ALSO

C Library: *strspn(), strstr(), strtok()*

# _strtoi

Convert the digits in a character-string to an *int.*

## SYNOPSIS

```
int _strtoi(str, ptr, base)
    char     *str;
    char    **ptr;
    int       base;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert to an integer |
| <ptr> | The address of the *char* * to contain the address of the character which terminates the conversion, or *(char **)* NULL if none |
| <base> | The base of the digits |

### Returns

The value generated from the character-string

## DESCRIPTION

The **_strtoi** function converts the character-string referenced by *<str>* to an *int.* The **_strtoi** function considers the digits to be in the base specified by *<base>* and assigns the address of the character ending the conversion to the *char* * referenced by *<ptr>*. The character that ends the conversion is either the null-character terminating the string or the first character that was inconsistent with the base. If *<ptr>* is *(char **)* NULL, the **_strtoi** function does not make this assignment.

If the argument *<base>* is greater than 0 and less than or equal to 36, that value is the base of the digits in the character-string. (For bases between 11 and 36, the alphabetic characters **A** through **Z** inclusive, in lexicographic order, are the digits of the base. The **_strtoi** function considers lower-case characters to be the same as upper-case characters.) If the base is 0, the function examines the character-string to determine the base. If **0x** or **0x** follows the optional white-space and sign, the base is assumed to be 16. Otherwise, if **0** follows the optional white-space and sign, the base is assumed to be 8. Otherwise, the base is assumed to be 10. If the base is less than 0 or greater than 36, the base is assumed to be 10.

## NOTES

The **_strtoi** function ignores overflow conditions.

## SEE ALSO

C Library: *_atoh(), atoi(), atol(), _atoo(), _atos(), strtol()*

# strtok

Extract the next token from a character-string.

## SYNOPSIS

```
#include <string.h>
char *strtok(s1, s2)
     char     *s1;
     char     *s2;
```

### Arguments

| | |
|---|---|
| <s1> | The address of the character-string to search, or *(char *)* NULL |
| <s2> | The address of the character-string containing the token separators |

### Returns

The address of the first character of the next token, or *(char *)* NULL if none

## DESCRIPTION

If the argument *<s1>* is not *(char *)* NULL, the **strtok** function begins scanning with the first character in the character-string with the address *<s1>* for the first character that is not in the token-separator character-string with the address *<s2>*. Otherwise, the **strtok** function begins scanning at the continuation-address set by a previous call, if any. If the **strtok** function finds no characters that are not token-separators, the function sets the continuation-address to *(char *)* NULL and returns *(char *)* NULL as its result. Otherwise, it remembers the address of that character as the value the **strtok** function returns as its result and continues scanning, looking for the next character that is a token-separator.

If **strtok** finds a token-separator, it changes that character to a null-character ('\0'), and sets the continuation-address to that of the character following that token-separator. Otherwise, **strtok** sets the continuation-address to *(char *)* NULL. The function then returns the remembered address, the address of the token, as its result.

## NOTES

The **strtok** function always returns *(char *)* NULL if it is called with the first argument *(char *)* NULL and there is no continuation-address. There is no continuation address if the function has not been called with the first argument something other than *(char *)* NULL or the function returned *(char *)* NULL the last time it was called.

The separator string referenced by <*s2*> does not have to be the same string from one call to this function to another.

If the function returns as its result something other than *(char *)* NULL, that result always references a character-string (a null-terminated string of characters).

The include-file <*string.h*> defines the string-handling functions in the C library.

## SEE ALSO

C Library: *strchr(), strpbrk(), strrchr()*

# strtol

Convert the digits in a character-string to a *long*.

## SYNOPSIS

```
long strtol(str, ptr, base)
     char     *str;
     char    **ptr;
     int       base;
```

### Arguments

| | |
|---|---|
| <str> | The address of the character-string to convert to an integer |
| <ptr> | The address of the *char* * to contain the address of the character that terminates the conversion, or *(char **)* NULL if none |
| <base> | The base of the digits |

### Returns

The value generated from the character-string

## DESCRIPTION

The **strtol** function converts the character-string referenced by *<str>* to a *long*. The **strtol** function considers the digits to be in the base specified by *<base>* and assigns the address of the character ending the conversion to the *char* * referenced by *<ptr>*. The character that ends the conversion is either the null-character terminating the string or the first character that was inconsistent with the base. If *<ptr>* is *(char **)* NULL, the **strtol** function does not make the assignment.

If the argument *<base>* is greater than 0 and less than or equal to 36, that value is the base of the digits in the character-string. (For bases between 11 and 36, the alphabetic characters **A** through **Z** inclusive, in lexicographic order, are the digits of the base. The **strtol** function considers lower-case characters to be the same as upper-case characters.) If the base is 0, the function examines the character-string to determine the base. If **0x** or **0x** follows the optional white-space and sign, the base is assumed to be 16. If **0** follows the optional white-space and sign, the base is assumed to be 8. Otherwise, the base is assumed to be 10. If the base is less than 0 or greater than 36, the base is assumed to be 10.

## NOTES

The **strtol** function ignores overflow conditions.

## SEE ALSO

C Library: *_atoh(), atoi(), atol(), _atoo(), _atos(), _strtoi()*

# stty

Set the characteristics of an open character-device.

## SYNOPSIS

```
#include <errno.h>
#include <sys/sgtty.h>
int stty(fildes, buf)
     int           fildes;
     struct sgttyb *buf;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the open character-device |
| <buf> | The address of the structure to contain the new characteristics |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **stty** function changes the characteristics of the open character-device referenced by the file descriptor *<fildes>* and to those described by the data in the structure referenced by *<buf>*. The **stty** function returns zero as its result if it successfully changes the characteristics of the open character-device. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **stty** function fails if the file descriptor is out of range, does not reference an open file, or does not reference an open character-device.

The function call expects *<buf>* to be the address of a structure that is defined as:

```
struct sgttyb
{
     unsigned char  sg_flag;
     unsigned char  sg_delay;
     unsigned char  sg_kill;
     unsigned char  sg_erase;
     unsigned char  sg_speed;
     unsigned char  sg_prot;
};
```

The *sg_flag* bit-string describes the current mode of the terminal. The values in the bit-string are:

| | |
|---|---|
| RAW | 0x01 |
| ECHO | 0x02 |
| XTABS | 0x04 |
| LCASE | 0x08 |
| CRMOD | 0x10 |
| SCOPE | 0x20 |
| CBREAK | 0x40 |
| CNTRL | 0x80 |

If RAW is set, the operating system considers the character-device to be in raw mode. In raw mode, the operating system suspends all processing of input and output. If RAW is clear, the operating system considers the character-device to be in non-raw mode (sometimes called *cooked mode*). In *cooked* mode, the operating system processes characters dependent upon the setting of the other bits in the bit-string.

If ECHO is set, the operating system echoes characters read to the character-device. If ECHO is clear, the operating system does not echo characters to the device.

If XTABS is set, the operating system expands tab-characters to spaces during output operations so that the next character written to the device is written to a column number that is an even multiple of eight. If XTABS is clear, the operating system writes tab-characters to the character-device with no expansion. Tab-characters are defined by the system as 0x09 and are defined by the C compiler as '\t'.

The LCASE mode is ignored, and is replaced by the CAPS key on the keyboard.

If CRMOD is set, the operating system writes a line-feed character to the character-device after every carriage-return character written. If CRMOD is clear, the operating system disables this feature.

If SCOPE is set, the operating system writes a backspace-character (0x08), followed by a space-character, followed by another backspace-character to the character-device when a character-cancel character is read from the character-device. If SCOPE is clear, the operating system disables this feature.

If CBREAK is set, the operating system considers the terminal to be in single-character mode. In this mode, the operating system reads data from the device one character at a time, passing each character to the calling task. If CBREAK is clear, the operating system considers the terminal to be in line mode, where it reads data from the device one line at a time, passing data to the calling task when a terminator is read.

If CNTRL is set, the operating system ignores all characters read from the character-device that are outside of the range 0x20 through 0x7E inclusive, except for the line-terminator character (carriage return), the keyboard-interrupt character (control-'c'), the quit-interrupt character (control-'\'), the character-cancel character, the line-cancel character, and the output-stop and output-start characters if any.

The *sg_delay* bit-mask indicates which characters, if written to the character-device, cause the operating system to pause before writing another character to the character-device. The values in the *sg_delay* bit string are:

| | |
|---|---|
| DELNL | 0x03 |
| DELCR | 0x0C |
| DELTB | 0x10 |
| DELVT | 0x20 |
| DELFF | 0x20 |

All of these modes are ignored on the 4400 Series.

The *sg_kill* value defines line-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default line-cancel character is control-'x' (0x18).

The *sg_erase* value defines the character-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default character-cancel character is the backspace character (control-'h', 0x08).

The *sg_speed* bit-mask contains configuration information for the character-device. Not all hardware supports the dynamic changing of the configuration. The values in *sg_speed* for the various configurations are:

| | | | |
|---|---|---|---|
| DSystem Call | EVEN | 0x00 | 7 data bits, 2 stop bits, even parity |
| DSystem Call | ODD | 0x04 | 7 data bits, 2 stop bits, odd parity |
| DC Library | EVEN | 0x08 | 7 data bits, 1 stop bit, even parity |
| DC Library | ODD | 0x0C | 7 data bits, 1 stop bit, odd parity |
| DSystem Call | NONE | 0x10 | 8 data bits, 2 stop bits, no parity |
| DC Library | NONE | 0x14 | 8 data bits, 1 stop bit, no parity |
| DC Library | EVEN | 0x18 | 8 data bits, 1 stop bit, even parity |
| DC Library | ODD | 0x1C | 8 data bits, 1 stop bit, odd parity |
| CONFIG | | 0x1C | mask for extracting configuration information |

The *sg_prot* field defines the type of start-stop protocol expected by the operating system for the character-device, and contains the baud rate used by the character-device. The values defined in that bit-string defining the protocol are:

| | |
|---|---|
| ESC | 0x80 |
| OXON | 0x40 |
| ANY | 0x20 |
| TRANS | 0x10 |
| IXON | 0x08 |

If ESC is set, the operating system stops writing to the character-device when it reads an escape-character (0x1B) from the device. The operating system resumes writing to the character-device when it reads another escape-character from the device.

If OXON is set, the operating system stops writing to the character-device when it reads an xoff-character (0x13). The operating system resumes writing to the character-device when it reads an xon-character (0x11).

If ANY is set, the operating system uses any character read from the character-device as a substitute for the xon-character.

If TRANS is set, the operating system xon-xoff is transparent for raw mode (see the bit-string *sg_flag* discussion).

The IXON mode is ignored on the 4400 Series.

The baud rates are defined in the field as:

| | | |
|---|---|---|
| BAUD_RATE | 0x0F | baud-rate mask |
| B75 | 0x01 | 75 baud |
| B110 | 0x02 | 110 baud |
| B134 | 0x03 | 134.5 baud |
| B150 | 0x04 | 150 baud |
| B200 | 0x05 | 200 baud |
| B300 | 0x06 | 300 baud |
| B600 | 0x07 | 600 baud |
| B1200 | 0x08 | 1200 baud |
| B1800 | 0x09 | 1800 baud |
| B2400 | 0x0A | 2400 baud |
| B3600 | 0x0B | 3600 baud |
| B4800 | 0x0C | 4800 baud |
| B7200 | 0x0D | 7200 baud |
| B9600 | 0x0E | 9600 baud |
| B19200 | 0x0F | 19200 baud |

Not all hardware supports all of these baud rates and not all hardware allows the dynamic changing of baud rates.

The include-file <*sys/sgtty.h*> contains the structure and data definitions described above.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EINVAL | An argument to the function is invalid |
| ENOTTY | The file is not a character device |

## SEE ALSO

System Call: *creat(), dup(), dup2(), gtty(), open(), pipe()*

Command: **conset**

## sync

Update the file-system.

## SYNOPSIS

```
int sync()
```

### Arguments

None

### Returns

Zero

## DESCRIPTION

The **sync** function updates the file-system so that the media match the internal description of the file-system. The **sync** function always returns zero as its result.

## ERRORS REPORTED

None

## SEE ALSO

Command: **update**

## sys_errlist

This is a global table containing references to messages describing system error codes.

### SYNOPSIS

```
extern char *sys_errlist[];
```

### DESCRIPTION

The **sys_errlist** table contains references to character-strings describing the meaning of system error code values.

### NOTES

Before using a system error code as an index into this table, compare it against the global variable *sys_nerr*. If it is greater than or equal to that variable, do not use it as an index into the table. Sometimes system error codes get added to the system before the list of error messages gets updated.

The character-strings are not terminated by an end-of-line character. Naturally, since they are character-strings, the characters in the strings are terminated by a null-character.

### SEE ALSO

C Library: *errno, _ierrmsg(), perror(), sys_nerr*

## sys_nerr

The number of system error messages referenced by the global table "sys_errlist".

### SYNOPSIS

```
extern int sys_nerr;
```

### DESCRIPTION

The **sys_nerr** external variable contains the the number of system error messages referenced by the global table *sys_errlist*. Before using an error code as an index into that table, it should be checked against this variable as error codes may be added to the system before the associated message is added to the global table.

### NOTES

This variable is **-1** if the system error message table has not been initialized by **_ierrmsg**().

### SEE ALSO

C Library: *errno, perror(), sys_errlist*

# system

Issue a shell command.

## SYNOPSIS

```
int  system(string)
     char     *string;
```

### Arguments

<string>  The address of a character-string containing a shell command

The target buffer address

### Returns

Exit status of **/bin/shell**, or **-1** with **errno** set to the system error code

## DESCRIPTION

This function causes *<string>* to be passed to **/bin/shell** as input, as if the string had been typed as a command at a terminal. The process waits until the shell has completed, then returns the exit status of the shell.

## NOTES

None

## SEE ALSO

System Call: *vfork(), execl()*

Command: *shell*

# tan

Calculate the tangent of an angle.

## SYNOPSIS

```
#include <math.h>
double tan(r)
     double    r;
```

### Arguments

| | |
|---|---|
| <r> | The angle whose tangent is to be computed. The tangent of the angle <r>. |

### Returns

Zero

## DESCRIPTION

This function calculates the tangent of the angle <r>. The function interprets the value <r> as an angle expressed in radians. It returns the calculated value as its result.

The function detects an overflow error if the result is beyond the range of the data type .LG double. If the function detects an overflow error, it calls "matherr()", passing to it the address of a filled .LG struct exception structure. It sets the "type" entry to OVERFLOW, "name" to the address of the character-string "tan", and "arg1" to <r>.

If "matherr()" returns 0, the function sets "errno" to ERANGE. The return value, which is system-dependent, is given in the tables in Section 3. If "matherr()" returns something other than 0, the function returns as its result the "retval" entry in the structure whose address was passed to "matherr()".

## ERRORS REPORTED

None

## SEE ALSO

System Call: *atan(), atan2(), cos(), matherr(), sin()*

## tanh

Calculate the hyperbolic tangent of an angle.

## SYNOPSIS

```
#include <math.h>
double tanh(x)
     double    x;
```

### Arguments

| | |
|---|---|
| <x> | The angle whose hyperbolic tangent is to be computed. The tangent of the angle <x>. |

### Returns

Zero

## DESCRIPTION

This function calculates the hyperbolic tangent of the value <x>. The hyperbolic tangent of <x> is defined as sinh(<x>)/cosh(<x>). It returns that value as its result.

## ERRORS REPORTED

None

## SEE ALSO

System Call: *cosh(), exp(), matherr(), sinh()*

# time

Get the current system-time value.

## SYNOPSIS

```
long time(ptime)
     long     *ptime;
```

### Arguments

<ptime>    The address of the *long* to receive the system-time value or *(long *)* NULL

### Returns

The current system-time value.

## DESCRIPTION

The **time** function gets the current system-time value (the current time-of-day) from the operating system and returns that value as its result. If *<ptime>* is not *(long *)* NULL, the **time** function stores the system-time value at the location referenced by *<ptime>*.

## ERRORS REPORTED

None

## NOTES

The operating system represents time as the number of seconds that has elapsed since the epoch. It defines the epoch as 00:00 (midnight) January 1, 1980 Greenwich Mean Time.

## SEE ALSO

System Call: *stime(), times()*

Command: **date**

# times

Get the CPU-usage information for the current task.

## SYNOPSIS

```
#include <sys/times.h>
int times(ptimes)
     struct tms    *ptimes;
```

### Arguments

| | |
|---|---|
| <ptimes> | The address of the structure to receive the task's current CPU-usage information |

### Returns

Zero

## DESCRIPTION

The **times** function gets the current task's CPU-usage information and places that information in the structure with the address *<ptimes>*. The CPU-usage information includes measurements of the Central Processing Unit (CPU) use for the task, the operating system CPU use on behalf of the task, the total CPU use of the children of the task, and total operating system CPU-use on behalf of the children of the task. The system measures CPU use in hundredths of a second. The **times** function always returns zero as its result.

The **times** function expects *<ptimes>* to be the address of a structure that is defined as:

```
struct tms
{
     long       tms_utime;
     long       tms_stime;
     long       tms_cutime;
     long       tms_cstime;
};
```

The *tms_utime* value contains CPU-time used by the current task. *tms_stime* contains the CPU-time used by the operating system in behalf of the current task. *tms_cutime* contains the total CPU-time used by all of the descendants of the task that have terminated. *tms_cstime* contains the total CPU-time used by the operating system in behalf of all of the descendants of the task that have terminated. The include-file *<sys/times.h>* defines this structure.

## ERRORS REPORTED

None

## NOTES

The operating system updates the CPU-usage information of the task for its descendants when a direct descendant task terminates. The operating system continuously updates the task's CPU-usage information for itself.

## SEE ALSO

System Call: *fork(), time(), vfork()*

Command: **shell**

# timezone

Current time zone value.

## SYNOPSIS

```
#include <time.h>
\xtern long timezone;
```

## DESCRIPTION

The **timezone** variable contains the current time zone value that is the number of seconds the zone is west of (behind) Greenwich Mean Time (Universal Coordinated Time).

This variable is initialized automatically by **localtime()** and **ctime()** and may be initialized explicitly by **tzset()**. Before initialization, the value is zero.

## NOTES

The include-file *<time.h>* defines this external variable along with other external variables and functions.

## SEE ALSO

C Library: *ctime(), daylight, localtime(), tzname, tzset()*

# toascii

Generate a value that is within the range of valid ASCII characters.

## SYNOPSIS

```
#include <ctype.h>
int toascii(c)
int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to be examined |

### Returns

<c> & 0x7F

## DESCRIPTION

The **toascii** function generates from the value *<c>* a value that is in the range of ASCII characters and returns that value as its result. It does this by anding *<c>* with the bit-string 0x7F (127). The result is a value between 0x00 and 0x7F inclusive, which is the range of ASCII characters.

## NOTES

The argument *<c>* is cast into an *int* if it is not already of that type. The include-file *<ctype.h>* defines this function and other functions that test and manipulate characters. The include-file *<ctype.h>* must be included in the C source before the first reference to the **toascii** function.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), _tolower(), toupper(), _toupper()*

# _tolower

Convert an upper-case character to a lower-case character.

## SYNOPSIS

```
#include <ctype.h>
int _tolower(c)
int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to convert |

### Returns

The converted value

## DESCRIPTION

The **tolower** function converts an uppercase alphabetic ASCII character to its equivalent lowercase alphabetic character and returns that value as its result.

## NOTES

This function is implemented as a macro. It has no side-effects but the result of the function is defined only for values of *<c>* that are upper-case alphabetic ASCII characters. The argument *<c>* is cast into an *int* if it is not already of that type. The include-file *<ctype.h>* defines this function and other functions that test and manipulate characters. It must be included in the C source program before the first reference to this function.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), toupper(), _toupper()*

## _toupper

Convert a lower-case character to an upper-case character.

## SYNOPSIS

```
#include <ctype.h>
int _toupper(c)
int  c;
```

### Arguments

| | |
|---|---|
| <c> | The value to convert |

### Returns

The converted value

## DESCRIPTION

The **_toupper** function converts a lowercase alphabetic ASCII character to its equivalent uppercase alphabetic character and returns that value as its result.

## NOTES

The **_toupper** function is implemented as a macro. It has no side-effects but its result is only defined for values of *<c>* that are lowercase alphabetic ASCII characters. The argument *<c>* is cast into an *int* if it is not already of that type. The include-file *<ctype.h>* defines this function and other functions that test and manipulate characters. The include-file *<ctype.h>* must be included in the C source before the first reference to the **_toupper** function.

## SEE ALSO

C Library: *isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper()*

# truncf

Set the size of an open file.

## SYNOPSIS

```
#include <errno.h>
int truncf(fildes)
    int       fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the file whose size is to change |

### Returns

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **truncf** function sets the size of the open file referenced by the file descriptor *<fildes>* so that its end-of-file is the current file position. If that position is before the existing end-of-file, **truncf** truncates the file. If that position is beyond the existing end-of-file, **truncf** extends the file. The function returns zero as its result if it successfully sets the size of the specified file. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The *truncf* function fails if the file descriptor *<fildes>* is not a valid file descriptor, does not reference an open file, or does not reference a file that has been opened for writing.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The file descriptor references a file that is not open for writing |
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EINVAL | An argument to the function is invalid |

## NOTES

If the **truncf** function truncates the file, all data beyond the new end-of-file is lost.

If the **truncf** function extends the file, it does so without allocating to the file any blocks of the medium where the file resides. Functions reading from the extended space read zeros.

## SEE ALSO

System Call: *creat(), dup(), dup2(), open(), pipe()*

# ttyname

Generate the pathname for a terminal.

## SYNOPSIS

```
        int        fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the terminal |

### Returns

The address of a character-string containing a pathname for the terminal or *(char *) NULL* if *<fildes>* is not a file descriptor for a terminal residing in the directory */dev.*

## DESCRIPTION

The **ttyname** function determines if the file referenced by *<fildes>* is a character-special file (a terminal), and is reached by a pathname that is the directory */dev.* If the file satisfies these conditions, **ttyname** generates a complete pathname for the file and returns as its result the address of a character-string containing that complete pathname. Otherwise, **ttyname** returns as its result *(char *)* NULL.

## NOTES

The character-string addressed by the result of the **ttyname** function is in static memory and is overwritten by subsequent calls to **ttyname**. A file descriptor is an index into the open file table of the operating system. The system functions **creat()**, **dup()**, **dup2()**, **open()**, and **pipe()** return a file descriptor as their result. The **fileno()** function determines the file descriptor of a stream.

## SEE ALSO

C Library: *fileno(), isatty()*

System Call: *creat(), dup(), dup2(), open(), pipe(), ttyslot()*

# ttyslot

Get the terminal number of the controlling terminal for the task.

## SYNOPSIS

```
int ttyslot()
```

### Arguments

None

### Returns

The terminal number of the controlling terminal for the task or zero if none

## DESCRIPTION

The **ttyslot** function gets the terminal number of the controlling terminal for the task and returns that value as its result. If the task has no controlling terminal, **ttyslot** returns zero as its result.

## ERRORS REPORTED

None

## NOTES

The operating system detaches a task from its controlling terminal if the task has its standard input file, its standard output file, and its standard error file closed simultaneously.

## SEE ALSO

C Library: *ttyname()*

## tzname

Time-zone name abbreviations.

### SYNOPSIS

```
#include <time.h>
\xtern char *tzname[2];
```

### DESCRIPTION

The **tzname** external variable is two-element array of references to character-strings. The first element references the three-character abbreviated name of the standard-time time zone, contained in a character-string. The second references the three-character abbreviated name of the daylight-time time zone, contained in a character-string. A (*char* *) NULL value indicates that **tzname** has not been initialized. A null-string indicates that the abbreviated name is not known.

The list is initialized automatically by **localtime()** and **ctime()** and may be initialized explicitly by **tzset()**. The values in the list are (*char* *) NULL before initialization.

### SEE ALSO

C Library: *ctime(), daylight, localtime(), timezone, tzset()*

## tzset

Initialize external variables containing time parameters.

## SYNOPSIS

```
void tzset()
```

### Arguments

None

### Returns

Void

## DESCRIPTION

The **tzset** function initializes the global variables **daylight, timezone**, and **tzname** according to the current system configuration.

The **daylight** variable is non-zero if the standard U.S.A. daylight-savings time conversion is applied to all time conversions from system time or Greenwich Mean Time (GMT) to the time in the local time zone, otherwise it is zero. The **timezone** variable contains the number of seconds the current time zone is west of GMT. The **tzname** array contains two elements, the first is the address of a character-string containing the abbreviation for the current standard time-zone name, the second is the address of a character-string containing the abbreviation for the current daylight time-zone name. If one or the other is not known, the strings are null-strings.

## NOTES

The **tzset** function is called automatically by **localtime()** and **ctime()**.

## SEE ALSO

C Library: *ctime(), daylight, localtime(), timezone, tzname*

# umask

Change the file-creation permissions mask for the task.

## SYNOPSIS

```
int umask(perms)
    int        perms;
```

### Arguments

| | |
|---|---|
| <perms> | A bit-string containing the new file-creation permissions mask |

### RETURNS

The previous value of the file-creation permissions mask

## DESCRIPTION

The **umask** function changes the file-creation permissions mask for the task to the low-order six bits of the bit-string *<perms>*. The **umask** function returns as its result the previous value of the file-creation permissions mask for the task.

The file-creation permissions mask describes the permissions that may not be applied to a created file. The file-creation function, **creat()**, ands the one´s-complement of the file-creation permissions mask of the task with the bit-string describing the permissions for the file being created, and applies the resulting permissions bit-string to the created file.

## ERRORS REPORTED

None

## NOTES

A task inherits its file-creation permissions mask from its parent.

## SEE ALSO

System Call: *creat(), fork(), fstat(), stat(), vfork()*

## umount

Unmount a mounted device.

## SYNOPSIS

```
#include <errno.h>
int umount(pathnam)
    char     *pathnam;
```

### Arguments

<pathnam> A character-string containing a pathname to the device to unmount

### RETURNS

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **umount** function unmounts the mounted device reached by the pathname in the character-string referenced by *<pathnam>*. The function returns zero if it successfully unmounts the device, Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **umount** function fails if the path in *<pathnam>* cannot be followed or contains a file which is not a directory. The **umount** function also fails the pathname does not reach a file, the file it reaches is not a device, the device is not mounted, or the device is busy.

## ERRORS REPORTED

| | |
|---|---|
| EBDEV | *<pathnam>* reaches something other than a device |
| EBUSY | The device is busy |
| EMSDR | The path in *<pathnam>* cannot be followed |
| ENMNT | The specified device is not mounted |
| ENOENT | *pathnam* does not reach a device |

## NOTES

A device that was mounted for read and write access but was not unmounted correctly cannot be mounted again until it is repaired by */etc/diskrepair*.

A device-busy error (EBUSY) usually indicates that a file on the specified device is currently open or that a task has as its working directory a directory on the device.

## SEE ALSO

C Library: *addmount(), rmvmount()*

System Call: *mount()*

Command: **/etc/diskrepair, /etc/mount, /etc/unmount**

# ungetc

Push a character onto an input stream.

## SYNOPSIS

```
int ungetc(c, stream)
int        c;
LE    *stream;
```

### Arguments

| | |
|---|---|
| <c> | The character to push onto the stream |
| <stream> | The stream to get the character |

### Returns

The argument *<c>* or *EOF*

## DESCRIPTION

If *<c>* does not equal EOF, the **ungetc** function pushes *(char) <c>* onto the standard I/O input stream *<stream>*. If **ungetc** succeeds, it returns its argument *<c>*, otherwise it returns *EOF*.

## NOTES

A stream may only have one character pushed onto it at a time. Attempting an **ungetc()** on a stream that already has a character pushed onto it results in losing the previously pushed character. The function returns *EOF* of the stream referenced by *<stream>* is not an input stream. The **fseek()** and **rewind()** functions undo the effects of this function. The result of **ftell()** does not reflect any character pushed onto the stream.

## SEE ALSO

C Library: *fdopen(), fgetc(), fopen(), fseek(), getc(), rewind()*

# unlink

Remove a link to a file.

## SYNOPSIS

```
#include <errno.h>
int unlink(pathnam)
     char      *pathnam;
```

### Arguments

<pathnam> The address of a character-string containing the pathname of the link to be removed

### RETURNS

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **unlink** function removes the pathname in the character-string referenced by *<pathnam>*. If that pathname is the last that reaches the associated file, the operating system deletes the file. The **unlink** function returns zero as its result if it successfully removes the pathname. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **unlink** function fails if the path in *<pathnam>* cannot be followed or it contains a file which is not a directory. The **unlink** function also fails if the pathname does not exist, or the directory containing the pathname does not grant the current effective user writing permission.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The directory containing the specified file does not grant writing permission to the current effective user |
| EMSDR | Could not follow the path to the file |
| ENOENT | The pathname does not reach a file |
| ENOTDIR | A part of the path is not a directory |

## NOTES

The **unlink** function can remove any entry in any directory that grants writing permission to the current effective user. That entry can be a directory that is not empty and can be the directories **.** and **..**.

If the *unlink* function removes the pathname to an open file, the operating system postpones deleting that file until it is closed.

## SEE ALSO

System Call: *creat(), link()*

Command: **create, link, remove, rename**

# urec

Remove an entry from the operating system lock table.

## SYNOPSIS

```
#include <errno.h>
int urec(fildes)
    int       fildes;
```

### Arguments

| | |
|---|---|
| <fildes> | The file descriptor with the lock table entry the **urec** function is to remove |

### RETURNS

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **urec** function removes from the operating system lock table the task's entry for the file referenced by *<fildes>*. The function returns zero if it successfully removes the entry, otherwise **-1** with *<errno>* set to the system error code.

The **urec** function fails if *<fildes>* is not a valid file descriptor or does not reference an open file.

## ERRORS REPORTED

| | |
|---|---|
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EINVAL | An argument to the function is invalid |

## NOTES

The **urec** function returns zero as its result if there is no entry in the operating system lock table for the specified file descriptor.

The operating system permits only one lock per file descriptor for each task.

Placing a lock on a portion of a file stops other tasks from placing a lock on the same portion of that file. It does not stop reading from and writing to that portion of the file.

## SEE ALSO

System Call: *lrec(), read(), write()*

# utime

Change the last-modification time for a file.

## SYNOPSIS

```
#include <errno.h>
#include <types.h>
int utime(pathnam, ptime)
     char     *pathnam;
     struct   *utime;
```

### Arguments

| | |
|---|---|
| <pathnam> | The address of a character-string containing a pathname for the file whose modification time is to change |
| <ptime> | The address of the the value to set as the file´s modification time |

### RETURNS

Zero if successful, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **utime** function changes the last-modification time for the file reached by *<pathnam>* to the system-time value referenced by *<ptime>*. The **utime** function requires that the current effective user be the system manager. The function returns zero if it successfully changes the file´s modification time, otherwise the function returns **-1** with *<errno>* set to the system error code.

The **utime** function fails if *<pathnam>* cannot be followed or *<pathnam>* contains a file that is not a directory. The **utime** function also fails if the pathname does not exist, the file reached by the pathname is currently open, or the current effective user is not the system manager.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The current effective user is not the system manager |
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EBUSY | The specified file is currently open |
| EINVAL | An argument to the function is invalid" |
| ENOENT | The pathname does not reach a file |

## NOTES

The operating system measures time as a count of seconds since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

The **utime** function does not compare the new modification time with the creation time or the current time, so it is possible to set the modification time of the file to before it´s creation date or to some time in the future.

## SEE ALSO

System Call: *fstat(), set_ftm(), stat()*

Command: **dir**

## vfork

Create a new task.

## SYNOPSIS

```
#include <errno.h>
int vfork()
```

### Arguments

None

### Returns

If successful, the child's task-ID to the parent (calling) task and zero to the child (created) task, otherwise **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **vfork** function creates a new task (the child task) that is an exact copy of the current task (the parent task). If the function succeeds, it returns the task-ID of the child to the parent task and returns zero to the child task. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **vfork** function fails if the current user can not allocate another task, the system task table is full, or if the current task may not call this function.

The **vfork** function differs from the **fork()** function call in that it generates the new task more efficiently on a virtual memory system. Instead of making a copy of the data in the parents memory, the child task inherits the memory allocated to the parent task. The child task is not allowed to call the **fork()**, **memman()**, or **vfork()** functions or any function that may change the memory configuration such as **sbrk()** or **stack()** until after it executes a program using the **execl()**, **execlp()**, **execv()**, or **execvp()** functions. The **vfork** function does not return to the parent task until the child task terminates or executes a program.

The child task is identical to the parent task in that it has the same task priority, user-ID, effective user-ID, controlling terminal information, file-creation permissions mask, working directory, signal handling set-up, profiling information, and allocated memory.

The child task differs from the parent task in that its task-ID is different, its parent task-ID is the task-ID of the parent task, its file descriptors are exact copies of those in the parent task, and its system and user CPU times are reset to zero.

## ERRORS REPORTED

| | |
|---|---|
| EAGAIN | The maximum number of tasks for the user are active or there are no available entries in the system task table |
| EVFORK | The task shares its memory with its parent and may not call this function |

## NOTES

A task-ID is a non-negative integer.

This function is the same function as **fork()** on systems that are not virtual memory systems.

The child task shares its stack with its parent so it should not return from the scope which calls the **vfork()** function.

The scope that calls the **vfork()** function must not have any register variables defined in it.

## SEE ALSO

C Library: *exit(), _exit()*

System Call: *execl(), execlp(), execv(), execvp(),fork(), memman(), wait()*

## wait

Suspend the task until a child task terminates.

## SYNOPSIS

```
#include <errno.h>
int wait(ptaskid)
     int      *ptaskid;
```

### Arguments

| | |
|---|---|
| <ptaskid> | The address of the *int* to get the termination status of the child task that terminated or *(int *)* NULL |

### Returns

The task-ID of the terminated task or **-1** with *<errno>* set to the system error code

## DESCRIPTION

The **wait** function suspends the current task until a child task terminates. When a child task terminates, if *<ptaskid>* is not *(int *)* NULL, the **wait** function puts the termination status of the terminated child into the value referenced by *<ptaskid>* and returns as its result the task-ID of the terminated child task.

If the function returns **-1**, it did not wait for a child task to terminate. The function returns without a child task terminating if there are no active child tasks or the task catches a signal.

The termination status contains the child task's exit code, the signal number that caused its termination, and a flag that indicates that it produced a core image file (a core dump). Anding the termination status with 0xFF00 extracts from it the child task's exit code. This is the high-order 8-bits of the argument to **exit()** (or **_exit()**) that terminated the task and typically indicates an error if it is not zero. Anding the termination status with 0x007F extracts from it the signal number that caused its termination. This is only non-zero if the child task did not terminate using the **exit()** or **_exit()** functions. Anding the termination status with 0x0080 extracts from it the core-image flag. If the flag is not zero, the child task produced a core image file when it terminated. Otherwise, it did not produce a core image file.

## ERRORS REPORTED

| | |
|---|---|
| ECHILD | There are no child tasks active |
| EINTR | The task caught a signal and that caused this function to end abnormally |

## SEE ALSO

C Library: *sleep()*

System Call: *alarm(), fork(), kill(), pause(), signal()*

Command: **int, wait**

# write

Write data to an open file.

## SYNOPSIS

```
#include <errno.h>
int write(fildes, bufad, nbytes)
     int       fildes;
     char     *bufad;
     int       nbytes;
```

### Arguments

| | |
|---|---|
| <fildes> | A file descriptor for the open file where the data is to be written |
| <bufad> | The address of the buffer containing the data to write |
| <nbytes> | The number of bytes of data to write |

### Returns

The number of bytes of data written to the file or **-1** if none with *<errno>* set to the system error code

## DESCRIPTION

The **write** function writes data to the file referenced by *<fildes>*. The **write** function writes data to the file from the buffer whose address is *<bufad>* and it writes at most *<nbytes>* bytes of data. If the **write** function successfully writes the data, it returns as its result the number of bytes of data that it wrote. Otherwise, it returns **-1** with *<errno>* set to the system error code.

The **write** function fails if the file descriptor is out of range, references a file that is not open for writing, or references a broken pipe. The function also fails if the disk is full or the operating system reports an I/O error while writing the data to the media. The **write** function may write less data than requested if it is writing to a slow device such as a terminal and the task catches a signal.

## ERRORS REPORTED

| | |
|---|---|
| EACCES | The file descriptor references a file that is not open for writing |
| EBADF | The file descriptor does not reference an open file or the file is not open in the proper mode |
| EINTR | The task caught a signal and that caused this function to end abnormally |
| EINVAL | An argument to the function is invalid |
| EIO | The operating system reports an I/O error |
| ENOSPC | The device is full |
| EPIPE | Attempting to write to a broken pipe |

## NOTES

This operation is most efficient when <*bufad*> and <*nbytes*> are evenly divisible by 512.

A broken pipe is one that has been closed for reading.

## SEE ALSO

C Library: *fwrite()*

System Call: *creat(), dup(), dup2(), open(), pipe(), read()*

# Section 4
# Graphics Library Concepts

## The Graphics Library

The graphics library is a library of C and assembly language callable functions. With these functions, you can easily display images on the 4400 Series machines´ displays. In addition to this, the library includes functions that allow you to use the mouse and keyboard as input devices in an application program. You can also clear the screen, turn the cursor and joydisk on and off, enter and exit the terminal emulator, save and restore the display "environment", enable and disable event processing in the operating system, and so forth. In short, in the graphics library, you have what you need to build sophisticated, graphics-oriented programs in C and assembly language using the graphics "building blocks" functions in the graphics library.

The presentation in this section concentrates on using the graphics library in C language programs. However, you will also find a discussion of how to call the library functions in assembly language, and, in addition, you will find one example of a simple assembly language program using library functions. Be sure to refer to the 4400 Series Assembly Language Programming Reference Manual for more information about assembly language programming in general.

## About This Section

The graphics library documentation has two basic parts:

- Section 4. A concept-oriented section, in which BitBlt graphics, the graphics environment, and operating system event processes are described as they relate to the graphics library.
- Section 5. A reference section, in which each function in the library is described succinctly for easy reference.

If you are acquainted with BitBlt graphics, you can safely skim Section 4 and use Section 5 as you need to during programming. On the other hand, if you have not done any graphics programming, you should take a closer look at Section 4. In Section 4, you will find a presentation of the concepts that lie behind the graphics functions. This will give you a good start on what you need to know to develop C graphics programs on a 4400 Series machine.

## Using the Graphics Library in C Programs

Your access to the graphics library is through two files: /lib/graphics and /lib/graphics.h. The /lib/graphics file is a library file that contains all the graphics functions documented here. To use this graphics library, you must, of course, tell the 4400 Series C Language compiler, cc, where to look for the graphics library. You can do this by typing this command at the operating system prompt:

```
cc <application-file-name>.c +l=/lib/graphics
```

("<application-file-name>" stands for the name of your application program.)

The /lib/graphics.h file in the /lib directory is a C language "include" file. This contains #define statements and structure declarations for the graphics data structures. You should always include the /lib/graphics.h file in your application programs. Here is an example:

```
/*
 * box.c — Draw a box in the middle of the screen with an "x"
 *       through the center.
 */
#include "graphics.h"
#define SIZE  50

main(argc,argv)
int argc;
char *argv[];
{

... C language declarations and statements ...

}
```

## Using the Graphics Library in Assembly Language Programs

Your access to the graphics library is through the file */lib/graphics*. To use the graphics library, you must tell the 4400 series loader, invoked with the *load* command, where to look for the graphics library. You can do this by typing the following command at the operating system prompt:

*load <program-file-name>.r /lib/Cwrapper.r+l=/lib/graphics +l=/lib/clibs*
("<program-file-name>" stands for the name of your assembly language program.)

Each of the graphics library functions can be called from an assembly language program by adding an underscore in front of the function name to indicate that it is a library function call. For example, *RectDrawX* is referenced as *_RectDrawX* in an assembly language program. Also, you need to declare each graphics library function you use as external, using the *extern* directive before it is referenced.

Parameters should be pushed onto the stack in reverse of the order indicated in the arguments list. For example, the library function *RectDrawX* expects two arguments: *rect* and *bbcom*. You need to first push the address of the *bbcom* structure onto the stack, then the address for the *rect* structure before you do a *jsr _RectDrawX* to call the *RectDrawX* function. The return code is always in *d0* and the condition codes are also set.

Here is an example:

```
*
* rect.a — draws a rectangle with upper left corner at (200,50),
*           width = 150, and height = 100.
*
        name   rect.a
        text
        extern _BbcommDefault
        extern _RectDrawX

        global _main
_main
        pea    bbcom
        jsr    _BbcommDefault              ; BbcommDefault(&bbcom)

        lea    rect,a0
        move.w        #200,(a0)            ; rect.x = 200
        move.w        #50,(a0)             ; rect.y = 50
        move.w        #150(a0)             ; rect.w = 150
        move.w        #100,(a0)            ; rect.h = 100

        pea    bbcom
        pea    rect
        jsr    _RectDrawX           ; RectDrawX(&rect, &bbcom)

        sys    term                 ; terminate task

        data
rect    ds.w   4
bbcom   ds.w   17

        end
```

## Graphics Environment and Structures

### Entering and Exiting Graphics Mode

The 4400 Series machines are similar to many *graphics terminals* on the market in the sense that the 4400 machines have a terminal mode and a graphics mode. Ordinarily, when you power up a 4400 machine, you load in the operating system which includes a terminal emulation module. You, as the user, issue commands to the operating system by means of the terminal emulator.

When you write graphics programs, you need to enter graphics mode. When you do this, you are actually mapping the bit-mapped display memory into the address space of the calling process in addition to putting the display into graphics mode. This is accomplished by calling the

InitGraphics function. With InitGraphics, you have the option of automatically invoking a number of additional modes or attributes over and above graphics mode. Some of these are: joydisk panning, clearing the screen, reversing the video, and so forth. You must, of course, invoke InitGraphics somewhere in the program before you start sending output to the display.

When you are done with the program and want to return the user to the operating system, you should invoke the ExitGraphics function to map the bitmapped display memory out of the process address space. Any graphics modes that have been enabled are unchanged by calling ExitGraphics.

## Environmental Settings

Under *Entering and Exiting Graphics Mode*, you learned about how to get into and out of graphics mode. But for a "cleanly" functioning graphics application program, this is not enough. As you are probably aware, at any given time, the 4400 Series display has a set of attributes associated with it – attributes such as whether joydisk and mouse action pan the display over the 4400 Series bitmap region, whether the mouse cursor is constrained to a certain portion of the display, whether the viewport itself is constrained to a certain portion of the display, whether the display is in normal video (black characters on white) or reverse video, and so forth. See the discussion below about DISPSTATE for more information.

### Saving and Restoring the Display State

You may decide as part of the programming functionality of your graphics application program that you want to save and restore these display attributes. You can do this with the SaveDisplayState and RestoreDisplayState functions. SaveDisplayState saves the current settings of many of the display attributes stored in the DISPSTATE structure. Invoking RestoreDisplayState restores the display attributes stored in a DISPSTATE structure by a previous invocation of saveDisplayState.

It is good programming practice to invoke these functions in the following order:

```
SaveDisplayState
InitGraphics
<<graphics code>>
ExitGraphics
RestoreDisplayState
```

Of course, after you have some acquaintance with these functions, you may choose to omit some of them. But, in the beginning, you will probably "decrease your learning curve time" by following the recommendation above.

### DISPSTATE

The DISPSTATE (display state) structure stores in one place all important attributes that affect the current environment of the display. These display attributes are saved and restored via the DISPSTATE structure. Most of the attributes are easily understood by reading the comments in graphics.h.

statebits is a long (32-bit) integer, each bit of which denotes one attribute that has one of two states – on or off. See the table for the definitions of each bit of statebits.

**Table 4-1**
*Statebits definitions*

| Bit | Definition and Values |
|---|---|
| 0 | 1=display enabled, 0=disabled |
| 1 | 1=screen attribute saving is enabled, 0=disabled |
| 2 | 1=video normal, 0=video inverse |
| 3 | 1=terminal emulator enabled, 0=disabled |
| 4 | 1=caps lock led on, 0=off |
| 5 thru 7 | reserved |
| 8 | 1=cursor visible enabled, 0=disabled |
| 9 | 1=cursor tracks mouse, 0=no tracking |
| 10 | 1=cursor panning enabled, 0=disabled |
| 11 | 1=joydisk panning enabled, 0=disabled |
| 12 thru 15 | reserved |
| 16 | 1=keyboard generates event codes, 0=not |
| 17 thru 31 | reserved |

## Panning

The PanDiskEnable and PanCursorEnable functions are simple functions that enable and disable joydisk panning and panning with the mouse (arrow) cursor.

## Mouse Bounds

The mouse cursor may be restricted to a certain part of the screen. You can specify the limits of motion with the SetMBounds function. You can also return the limits to your program with the GetMBounds function.

## Viewport

Sometimes you may want the 4400 Series screen to pan within a portion of the total screen bitmap in graphics memory. The (0,0) position of the display – the upper left corner pixel – is normally set to the (0,0) position in the graphics memory bitmap. You can alter this with the SetViewport function. You can also retrieve the current position of the upper left corner by invoking the GetViewport function.

## Cursors and Halftones

A selection of cursor forms and halftone forms are provided by the graphics and events Library. Cursors and halftones are special forms (see FORMS below) which are exactly 16 by 16 bits (pixels). Cursors may have a "hotspot" defined by the value of the offset fields. The standard cursors include NormalCursor, OriginCursor, CornerCursor, WaitCursor, and

CrosshairCursor. To set a particular cursor, use the SetCursor function.

```
SetCursor(&CornerCursor);
```

The standard halftone forms include WhiteMask, VeryLightGrayMask, LightGrayMask, GrayMask, DarkGrayMask, and BlackMask. A halftone form may be used as an option with BitBlt operations. (See *BitBlt Graphics* below).

### Screen Size Constants

When you use the function InitGraphics, the following values are set: ScrWidth, ScrHeight, ViewWidth, and ViewHeight. In order to write code that is portable across the 4400 Series, you should use these "externs" into display size storage rather than "hard" constants like 640 or 1024. Just remember to initialize graphics before referencing these variables – otherwise all are equal to zero.

## Graphics Structures

A number of C language structures have been defined that the graphics library functions use. These definitions are found in the file /lib/include/graphics and they are described here as an introduction to the BitBlt discussion later.

### POINT

POINT is a simple structure consisting of two short integers x and y. x is an x-axis value of the screen; and, thus, valid values are in the range of the 4400 Series machine screen bitmap. For the 4404 machine, this is 0 to 639. For other 4400 Series machines, this range may be larger. y is an y-axis value of the screen; and, thus, valid values are in the range of the 4400 Series machine bitmap. For the 4404 machine, this is 0 to 479. For other 4400 Series machines, this range may be larger.

Here is the POINT declaration from graphics.h:

```
struct POINT {
   short x, y;
};
```

### RECT

RECT, like POINT, is a simple structure. It consists of two short integers, x and y, and two other short integers, w and h. x and y denote the upper left corner point of a rectangular array of bits in a bitmap. w stands for the width in bits of a rectangular array of bits and h stands for the height in bits of a rectangular array of bits.

Here is the RECT declaration from graphics.h:

```
struct RECT {
   short x, y;
   short w, h;
};
```

## FORM

The FORM structure is used to locate and retrieve images intended for manipulation or display on the 4400 Series machine screen. These images reside in graphics memory. See Figure 4-1, *A 16 by 16 Bit Form.*

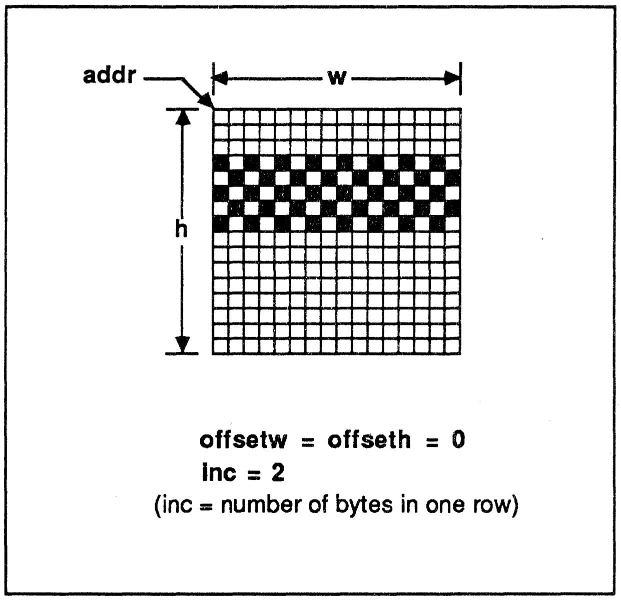


**Figure 4-1. A 16 by 16 Bit Form.**

A FORM structure consists of four short integer declarations:

- addr
- w and h
- offsetw and offseth
- inc

addr is a pointer that refers to the location in the graphics memory where the byte holding the upper left bit of a form is. A form is a rectangular array of bits that refers to a particular location in graphics memory. Forms are the objects that the BitBlt operation manipulates.

w and h are the short integers that represent the width and height in bits of a form.

offsetw and offseth are the short negative integers that may be specified if you desire to operate on a point within a form. These variables have been used with special forms called cursor forms to represent the active point within the interior of a cursor form. For example, the tip of the arrow cursor in the Smalltalk system is within a 16 by 16 pixel (bit) sized cursor form. The tip of the arrow is the active point (selection point) and the tip location in the form is denoted by the offsetw and offseth variables. In the graphics library, these variables are used by all functions related to cursor/mouse position.

inc is a short integer that denotes the number of bytes (always even) in one row of a form. This is a requirement for efficient use of graphics memory. The actual image width and height (w and h) in a form is possibly less than the number of bits, reserved in the number of even bytes specified by inc. Thus, inc reserves enough room in graphics memory for an image specified by appropriate w and h values.

Here is the FORM declaration from graphics.h:

```
struct FORM {
   short *addr;                    /* memory address of first bit of bitmap */
   short w, h;                     /* width and height of form */
   short offsetw, offseth;         /* initialized to zero, not used by library */
   short inc;                      /* byte increment from one line to next,
                                   must be even */
};
```

## FONT

The structure used for font information is called FontHeader. You must include the include file font.h with your program if you work with fonts.

A broad selection of standard system fonts is stored in the directory /fonts. (See **Graphical Text** later for more information.)

## MENU

The MENU structure is used for information about pop-up menus. This includes information necessary to display the menu and return a menu selection. (See **Menus** later for more information.)

## Bitblt Graphics

The graphics functions in the graphics library are built upon the BitBlt notion of doing computer graphics. The term "BitBlt" was coined by the creators of the Smalltalk language and environment. (The 4400 Series Artificial Intelligence Machines implement the version of this language and environment released in 1980.) The Smalltalk creators saw that a wide range of images could be efficiently displayed and updated on a raster scan display by defining and manipulating two rectangular matrices of bits called the source and destination forms. By letting a 1 bit be equivalent to a dark pixel on the screen and a 0 bit be equivalent to a light pixel, it is possible to build up a complex image in a rectangular source "form". If you define the destination form to be that part of system memory dedicated to the screen, then performing a

BitBlt operation, which transfers images from the source to the destination form, is equivalent to writing directly to the screen. This makes for fast update of images on the display.

Using two forms,,the source and destination forms, allows you to manipulate the image during the process of transferring it from the source to the destination form. A third form, called the halftone form, is sometimes combined with the source form before the source is transferred to the destination form. Halftoning accomplishes the "coloring" of the entire source form image with a (usually uniform) masking form, the halftone form.

When the source form is transferred to the destination form, a specific combination rule is used. There are sixteen possible rules, only one of which is used at a time. These rules are applied to the corresponding bits in the source and destination forms. For example, the corresponding bits of each form may be ANDed, or they may be XORed, or they may be ORed, etc.

Sometimes the entire source form image may *not* be needed on the display. In this case, a clipping rectangle is defined which effectively cuts off the unwanted portion of the source form image from being transferred to the destination form. The parts of the source form image positioned outside the clipping rectangle are not, of course, transferred to the destination form.

Thus, there are three basic ways to manipulate the source form image:

| | |
|---|---|
| halftoning | The source form is masked by a secondary "tinting" or "coloring" pattern. |
| combination rule | The source form corresponding image bits are combined one at a time with one of sixteen logical bit operations – for example, make the destination bits all ones, make the destination bits all zeros, AND the source and destination bits, XOR the source and destination bits, etc. |
| clipping rectangle | This device provides a way to selectively display portions of the source form image. |

## The BBCOM Structure

BBCOM stands for "BitBlt Command". BBCOM is a structure that stores or points to all of the data that the BitBlt operation needs to copy bits from one place to another, with possible manipulation of bit values during the copy operation. See Figure 4-2, *The BBCOM Data Structure.*

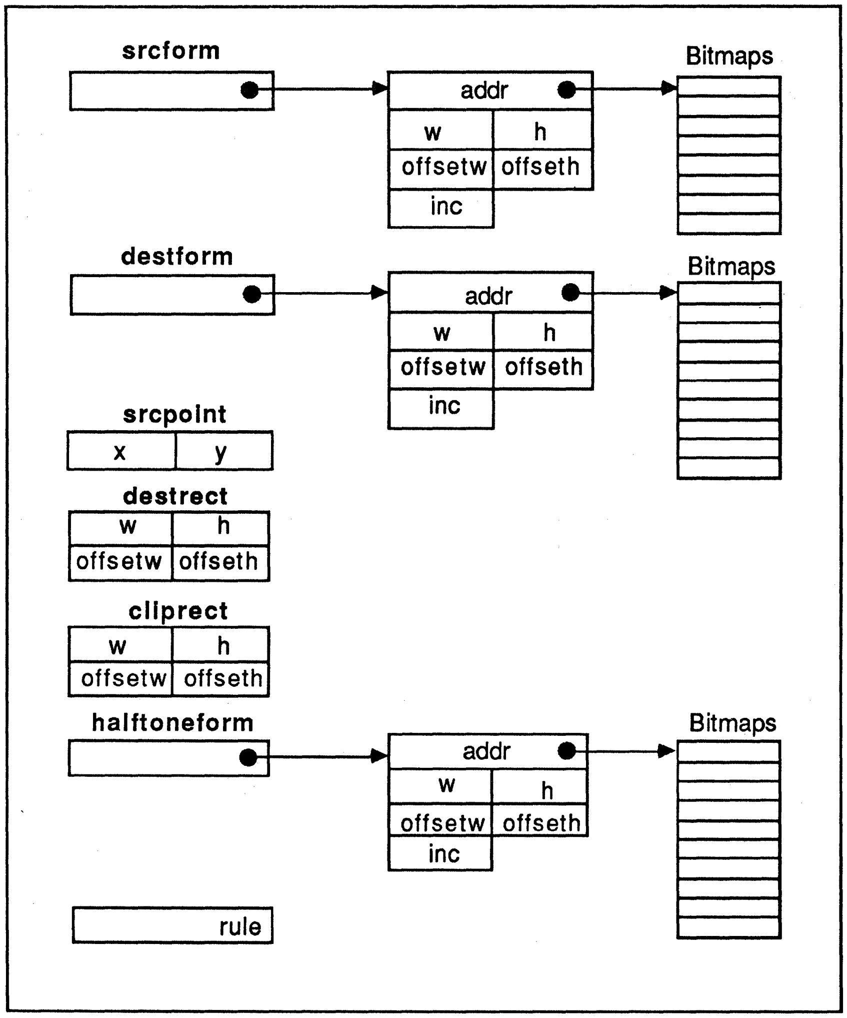


**Figure 4-2. The BBCOM Data Structure.**

Here is what the BBCOM structure consists of:

- srcform
- destform
- srcpoint
- destrect
- cliprect
- halftoneform
- rule

Note that some previously defined structures are used to define the BBCOM structure. In fact, of course, the RECT and FORM structures were defined primarily for the purpose of defining BBCOM in a conceptually straightforward way.

Here is the BBCOM declaration from graphics.h:

```
struct BBCOM {
   struct FORM  *srcform;          /* defines source form, NULL if not needed */
   struct FORM  *destform;         /* defines dest form */
   struct POINT srcpoint;          /* source coord, 0,0 is top left */
   struct RECT  destrect;          /* rectangle for use in dest bitmap */
   struct RECT  cliprect;          /* clipping rectangle */
   struct FORM  *halftoneform;     /* form for halftoning, NULL if not needed */
   short  rule;                    /* combination rule (defined below) */
};

/* Bitblt combination rules */
#define bbZero                     0  /* = zeros */
#define bbSandD                    1  /* = source and dest */
#define bbSandDn                   2  /* = source and dest' */
#define bbS                        3 /* = source */
#define bbSnandD                   4/* = source' and dest */
#define bbD                        5  /* = dest */
#define bbSxorD                    6 /* = source xor dest */
#define bbSorD                     7  /* = source or dest */
#define bbnSorD                    8  /* = (source or dest)' */
#define bbnSxorD                   9  /* = (source xor dest)' */
#define bbDn                       10 /* = dest' */
#define bbnD                       10 /* = dest' */
#define bbSorDn                    11 /* = source or dest' */
#define bbSn                       12 /* = source' */
#define bbnS                       12 /* = source' */
#define bbSnorD                    13 /* = source' or dest */
#define bbnSandD                   14 /* = (source and dest)' */
#define bbOnes                     15 /* = ones */
```

srcform is a pointer to a FORM structure. So, srcform points to a rectangular array of bits in graphics memory. The "source form" contains a predefined pattern of bits forming an image of some sort. This image may exist in main memory or as a file on disk. You can create a source form within the same program that reads it, or you may have created a form within another C graphics program that stored the image in a disk file. (See later in this section about how to create an image in the Smalltalk system and then write it to a file on disk.) Remember that the srcform has an addr, width and height, possibly an offsetw and offseth, and an inc. See under FORM above for the definitions of these variables.

destform is a pointer to a FORM structure. A destform may be a FORM defined to be in graphics memory only or it may be identified with the 4400 Series display screen. As its name implies, the destform becomes the destination form when the BitBlt operation is performed on the source form, which usually contains the image of interest.

srcpoint is a POINT structure that refers to a point within the source form array of bits. This point is the location of the upper left bit in the source form where the transfer (or copy) operation that is the BitBlt operation, begins. You can, of course, specify srcpoint to be (0,0) in which case you obtain a copy of all the bits in the source form to the destination form. Or, you can specify that srcpoint be an "embedded" point in which case you obtain a partial copy of the bits in source form.

destrect means "destination rectangle". Note that this is a RECT structure and not a FORM structure, and, as such, it has no connection with graphics memory; that is, it has no addr pointer. destrec is used to specify two parameters:

- Where in the destination form the BitBlt copy operation is to start copying bits. This may be at (0,0) or any other point in the destination form. This is specified by the x and y variables in the RECT structure.
- How much of the source form is copied to the source form. This is specified by the w and h variables in the RECT structure.

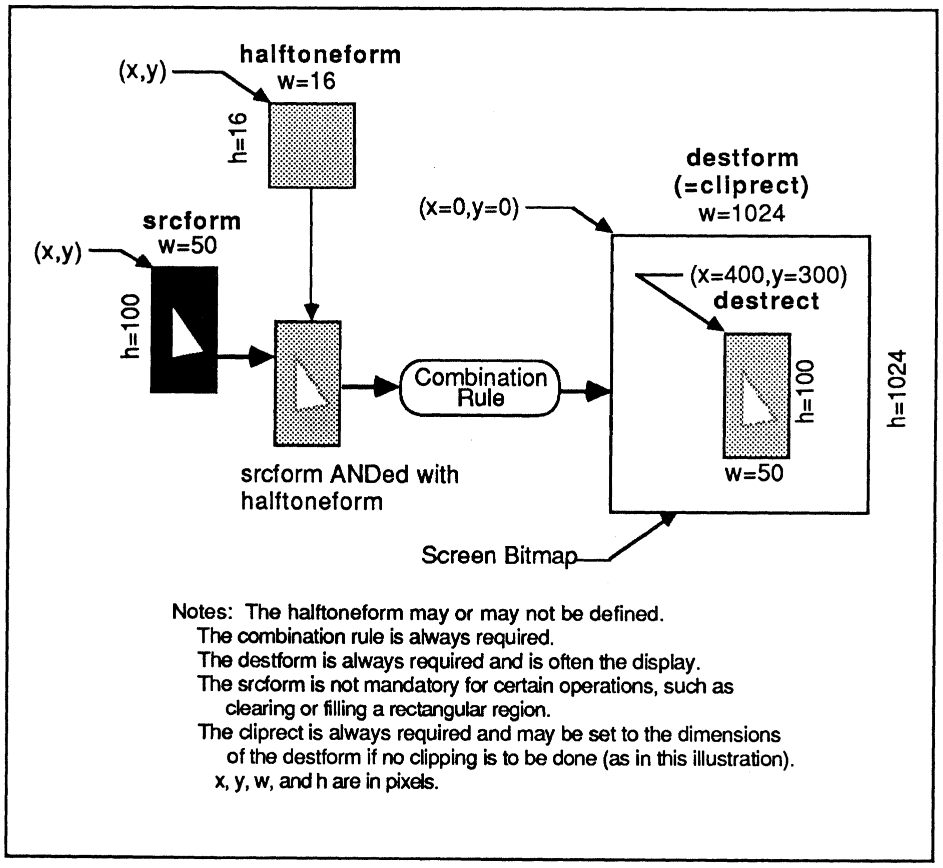
cliprect is another RECT structure and denotes the clipping rectangle. You can think of cliprect as a rectangular array of bits embedded in the destination form. The purpose of cliprect is to (possibly) exclude part of the destination form image from being displayed on the 4400 Series screen. Only those bits falling inside the clipping rectangle are displayed on the 4400 Series screen. If you do not desire any clipping to occur, then you simply define the clipping rectangle to have the same dimensions as the destination form or the 4400 Series screen.

halftoneform is a pointer to a special FORM structure that provides a way to "color" or "tint" the destination form as the source form is copied into it during the BitBlt operation. The 16 by 16 bit halftone form is a (usually) uniformly patterned array of bits meant to function as a mask or "halftone".

rule is a short integer that denotes one of sixteen different combination rules. These rules are based upon the logical connectives: and, or, xor, negation, etc. The combination rule operates on the corresponding bits in the source and destination forms during the BitBlt operation. Thus, for example, bit (x=16, y=33) in the source form is ANDed with bit (x=16, y=33) in the destination form. See the #define statements in the BBCOM above, for which integer to use for a particular combination rule. Some of the more useful rules are: bbZero, bbOnes, bbSxorD, bbSandD, bbSorD, bbS, and bbD. bbSxorD is especially useful since you can recover the original image by two applications of the rule, one immediately after the other. See later for the use of some of the other rules.

## How the BitBlt Operation Works

When your program calls the BitBlt function, it goes through a process something like the following. (Refer to Figure 4-3, *The BitBlt Operation.*) BitBlt first looks for a source form and a halftone form. If it finds both, it does an AND operation on the corresponding bits of the source form and the halftone form. Note that, as the figure indicates, the halftone form is actually a 16 by 16 bit square form. This means that the halftone is repeatedly ANDed with the source form until the entire source form has been ANDed with the appropriate number of halftone forms. If the halftone form is null, this part of BitBlt is not performed.



Notes: The halftoneform may or may not be defined.
The combination rule is always required.
The destform is always required and is often the display.
The srcform is not mandatory for certain operations, such as clearing or filling a rectangular region.
The cliprect is always required and may be set to the dimensions of the destform if no clipping is to be done (as in this illustration).
x, y, w, and h are in pixels.

**Figure 4-3. The BitBlt Operation.**

Next, BitBlt looks for the destination form that you have specified. (Many times this is identified with the special screen form.) Now the combination rule you have specified is applied to every corresponding source form bit and destination form bit. (When the destination form has been identified with the screen form, bits are equivalent to screen pixels.)

Also during this time, BitBlt looks for a clipping rectangle and destination rectangle you may have specified. If you have specified a clipping rectangle, BitBlt cuts off all those parts of the image that fall outside it in the destination form. And, if you have specified a destination rectangle, BitBlt starts copying bits beginning at the upper left corner of the destination rectangle. (You have most probably specified that the upper left corner of the destination *rectangle* be somewhere within the destination *form*. See Figure 4-3, *The BitBlt Operation.*)

As you can see, the BitBlt function has a lot of functionality which you control by a careful selection of the variables within the source form, destination form, combination rule, clipping rectangle, destination rectangle, and halftone form. Let's look at a simple example of the use of BitBlt.

## Drawing a Box on the Screen

This program uses the PaintLine function from the graphics library. PaintLine calls the BitBlt function internally, and, thus, uses the BBCOM data structure of BitBlt. However, PaintLine makes the drawing of lines on the screen more straightforward than BitBlt. That is why it is used here. See also LineDraw and LineDrawX for even simpler drawing of lines.

(BitBlt itself is used in other programs in the /samples directory. You may want to take a look at those as you read through this discussion.)

First, run the example program by typing at the system prompt:

```
/samples/box
```

You should see the screen clear and a box with two intersecting lines drawn through the center of the box. The program returns you to the operating system. Now, refer to the following text of the program while you read through the discussion.

```
/*
 * box.c — Draw a box in the middle of the screen with an "x"
 *        through the center.
 */
#include "graphics.h"
#define SIZE                              50

main(argc,argv)
int argc;
char *argv[];
{
   struct FORM *screen;
   struct BBCOM bbcom;
   struct POINT p;

   screen = InitGraphics(FALSE);
   bbcom.destform = screen;                 /* destination is screen */
   bbcom.srcform = (struct FORM *)NULL;     /* no source */
   bbcom.destrect.w = 1; /* one pixel wide line to draw with */
   bbcom.destrect.h = 1; /* one pixel wide line to draw with */
   bbcom.srcpoint.x = bbcom.srcpoint.y = 0;
   bbcom.cliprect.x = bbcom.cliprect.y = 0;
   bbcom.cliprect.w = ScrWidth;             /* clip to virtual screen */
   bbcom.cliprect.h = ScrHeight;
   bbcom.halftoneform = (struct FORM *)NULL;/* no halftone */
   bbcom.rule = bbOnes;                     /* black lines if video=normal */
```

```
        /* now draw the box */

    bbcom.destrect.x = bbcom.destrect.y = 50;  /* beginning point */
    p.x = 50; p.y = 430;                       /* set up the point PaintLine draws to */
    PaintLine(&bbcom, &p);                     /* draw the line */
    bbcom.destrect.x = 50; bbcom.destrect.y = 430;
    p.x = 590; p.y = 430;
    PaintLine(&bbcom, &p);
    bbcom.destrect.x = 590; bbcom.destrect.y = 430;
    p.x = 590; p.y = 50;
    PaintLine(&bbcom, &p);
    bbcom.destrect.x = 590; bbcom.destrect.y = 50;
    p.x = 50; p.y = 50;
    PaintLine(&bbcom, &p);

        /* draw the intersecting lines inside the box */

    bbcom.destrect.x = bbcom.destrect.y = 50;
    p.x = 590; p.y = 430;
    PaintLine(&bbcom, &p);
    bbcom.destrect.x = 590; bbcom.destrect.y = 50;
    p.x = 50; p.y = 430;
    PaintLine(&bbcom, &p);
}
```

Every program that uses the graphics library functions must include the graphics.h include file. Be sure to always "include" it.

## Initializing Graphics Mode

Note especially the struct FORM *screen declaration. This declaration sets up a FORM structure that becomes the screen bitmap. This happens via the line screen = InitGraphics(TRUE). InitGraphics not only puts the system into graphics mode but also returns a pointer to a form that defines the screen bitmap. The very next line, bbcom.destform = screen, assigns the screen to the destination form of the BitBlt data structure. This means, of course, that any changes made to the destination form are made directly on the 4400 Series machine screen.

## Setting Up the BitBlt Structure

In this application, the source form is not used, so it is assigned a NULL value. That is, there is no pre-existing form to be manipulated by BitBlt operations. You could have created this form within the application program or you could have had the program read in a previously created image that existed as a file on disk. See later in this section for a way to read in an image file created within the Smalltalk-80 system and then stored on disk.

The next two lines (bbcom.destrect.w = 1 and bbcom.destrect.h = 1) may not be easily figured out. These lines fix the width and height of the destination rectangle to a 1 by 1 rectangle, that is, a single pixel written into the destination form. So, the destination rectangle, when it is repeatedly written into the destination form, appears as a connected series of pixels on the screen. This gives you the line that you want to draw the lines of the box. (An interesting exercise is to vary the size of w and h. Try bbcom.destrect.w = 5 and bbcom.destrect.h = 5. You get "fatter" lines.)

The source points bbcom.srcpoint.x and bbcom.srcpoint.y are both set to zero here since you are not using the source form.

The clipping rectangle x and y values are set to zero. This means that the upper left corner of the clipping rectangle is coincident with the upper left corner of the destination form, which in this case is the screen bitmap. To see if there will be any clipping of the destination form image, look at the values for the clipping rectangle width and height – bbcom.cliprect.w and bbcom.cliprect.h. Look carefully at what the width and height are set to in this example. The clipping rectangle width and height are set to two graphics.h file variables that are declared as extern short integers; these are ScrWidth and ScrHeight. Thus, these variables, which are in effect constants, get their values outside the graphics.h file. But where? The answer is: ScrWidth and ScrHeight have their values set when you call the InitGraphics function. This is one of the reasons you should call InitGraphics early in your program. So, you have set the clipping rectangle to be the width and height of the screen bitmap. This implies that there is no clipping of the image in the destination form in this program.

The halftone form is not used, so it is set to NULL.

The combination rule here is set to bbOnes. If you have set the video to normal (black on a white background), then bbOnes gives you black lines when the box is drawn. With the combination rule, you have finally set all the values of the BitBlt structure. Note that even if a variable in the structure is not used, it is still set to a definite value – NULL or zero. This is good programming practice and helps prevent unexpected effects.

### Drawing the Box

The box with the x through it is drawn by specifying two points and then drawing a line between them. This is done six times – one for each line.

The PaintLine function needs two points. One of the points is the upper left corner of the destination rectangle in the BitBlt structure and the other point is defined for the Paintline function. This is point p.x, p.y. The point p takes values within the destination form, which in this program is confined to the screen bitmap. Note that the destination rectangle point is the beginning endpoint for Paintline and p is the ending endpoint. So, the action of Paintline is to start at the beginning endpoint and repeatedly call the BitBlt operation, which displays a sequence of adjacent pixels until it reaches the ending endpoint p. (Remember that the destination rectangle has been "collapsed" into a single pixel in this program since bbcom.destrect.w = bbcom.destrect.h = 1.)

## An Interesting Example Program

In the /samples directory, you will find a C graphics demonstration program by the name of tvcam.c. This is a simple program that still manages to show you the power of BitBlt.

Before you run the program, though, make sure that you have some text or other output on the screen. You can do this by running the vecdemo demonstration program also found in the

/samples directory. vecdemo runs through a sequence of vector demonstration images and then exits back to the operating system. You can either let vecdemo run to completion or control-c out of an interesting part of the sequence. The point is simply to leave an image of some sort on the screen so that the tvcam program has something to work on. Run vecdemo by typing this at the operating system prompt:

```
/samples/vecdemo
```

Now you can run tvcam to see how it works. Type this at the system prompt:

```
/samples/tvcam
```

Move the mouse rapidly around the screen and observe the center of the screen. You see that what appears in the "window" in the center of the screen is directly related to the position of the mouse arrow cursor. Specifically, imagine that the arrow cursor tip is the lower right corner of a square 100 by 100 pixels in size. Now watch the central "window" as this imaginary square moves over the screen, tracking with the arrow cursor. Whatever appears in the imaginary square is copied via BitBlt to the central "window". Watch especially what happens when you place the imaginary square near the central "window"!

Now look at the tvcam.c program show below. Note as always that graphics.h must be "included".

```
/*
 * tvcam.c — copy a rectangle from the mouse to a fixed place on the screen.
 *        This has some great opportunities for video feedback.
 */
#include "graphics.h"
#define SIZE                              100

main(argc,argv)
int argc;
char *argv[];
{
   struct FORM *screen;
   struct BBCOM bbcom;
   struct POINT cur;
   register int i, j, ret;

   screen = InitGraphics(FALSE);
   /* set up the bitblt command structure */
   bbcom.srcform = bbcom.destform = screen; /* source and dest are screen */
   bbcom.destrect.x = bbcom.destrect.y = 200;/* output position */
   bbcom.destrect.w = bbcom.destrect.h = SIZE;
   bbcom.cliprect.x = bbcom.cliprect.y = 0;      /* clip to virtual screen */
   bbcom.cliprect.w = ScrWidth;
   bbcom.cliprect.h = ScrHeight;

   bbcom.halftoneform = (struct FORM *)NULL;/* no halftone */
   bbcom.rule = bbS;                         /* rule Dest=Source */

   /* now turn on cursor and link it to mouse */
   CursorVisible(TRUE);
```

```
    CursorTrack(TRUE);

    /* now the loop */
    for (i = 0; i < 1000; i++) {
         GetCPosition(&cur);
         cur.x -= SIZE;
         cur.y -= SIZE;
         bbcom.srcpoint = cur;            /* copy point structure into bb command */
         ret = BitBlt(&bbcom);            /* copy from mouse to dest rect */
         if (ret != 0) {
            printf("BitBlt failed, returned %d0, ret);
         }
         for (j = 0; j < 5000; j++)
            ;
    }

    /* turn off and clean up */
    ClearScreen();
    cur.x = cur.y =0;
    SetCPosition(&cur);
    CursorTrack(FALSE);
    CursorVisible(FALSE);
}
```

A FORM structure is declared called screen and a POINT structure, called cur (cursor), is also declared. The next line of interest is screen = InitGraphics(FALSE). This line says to leave the display environment alone and especially to not clear the display. Also, it says to assign the pointer to the bitmap screen memory to the variable screen. Note carefully the next line. Here screen, bbcom.srcform, AND bbcom.destform are all assigned to the bitmap screen memory. So, in this program, when BitBlt operates, it copies pixels from one part of bitmap screen memory to another part of bitmap screen memory.

Next the destination rectangle is set up in the approximate middle of the screen and is made 100 by 100 pixels in size. The clipping rectangle is safely set the size of the bitmap screen memory, and, thus, has no effect here. The halftone form plays no part in this program is set to the NULL form. However, look at what the combination rule is set to. bbS means that the destination form pixels become whatever the corresponding source form pixels are.

The cursor is important in this program, so it is made visible and it is made to track with the mouse – CursorVisible(TRUE) and CursorTrack(TRUE). The program has two *for* loops – one inside the other. The inner loop is executed 5000 times and the outer loop 1000 times. The program then clears the screen, sets the cursor position to (0,0) – the upper left corner of the screen, turns off cursor tracking, and finally makes the arrow cursor invisible.

So what happens inside the loops? First, the current position of the cursor is returned in cur, then SIZE(=100) is subtracted from the x and y components of the point. Next, the cur point is made the source point in the source form. You will probably remember that the source point is the point at which the BitBlt operation starts its copy operation. Since the destination rectangle is 100 by 100 pixels in size, it appears as if BitBlt is copying from a 100 by 100 pixel-sized region in the source form, which here is the bitmap display memory, that is, the screen. And, thus, the mouse cursor appears to be at the lower left corner of an imaginary 100 by 100 pixel square.

Finally, the BitBlt function is called and operates on the bbcom structure defined for it. Note the use of the register variable, ret to provide an error report if BitBlt fails.

## Graphics Error Messages

Functions of Graphics and Events Library detect errors, but leave it up to the application program as to how an error should be handled. When an error is encountered, special values are returned, and a special variable called errno is set to an error code. Functions which normally return a pointer to a structure, return NULL if there has been an error. All other functions return a negative value (usually –1) to signify an error condition.

If you would like to print out a descriptive message, you can use the standard C function perror, as shown in the following program excerpt.

```
if (GetCursor(NULL) < 0) perror("Error in GetCursor");
```

The argument to GetCursor must be a pointer to a 16 by 16 bit form. This code segment would result in the following message directed to standard out.

```
Error in GetCursor: Invalid (NULL) structure pointer
```

You can also do more sophisticated error handling based on the value of the error code stored in errno. If you want to handle errors based on the value of errno, you must include the include file errno.h with your program.

## Creating Images In Forms

The Smalltalk bit editor may be used to create a form that can be read and used for graphics in C programs. (The following brief discussion shows you how to create a form in the Smalltalk-80 system that you got as a standard part of your 4400 Series machine. You should already have gone through the manual *Introduction to the Smalltalk-80 System* before you attempt to create the form in Smalltalk.)

To create the form newcursor.form, perform these steps:

1. At the operating system prompt, invoke smalltalk:

   ```
   smalltalk standardImage
   ```

2. Open up a workspace with the middle button menu when the cursor is on the background.
3. Enter the following smalltalk statements in the workspace.

   ```
   aForm ← Form new extent: 16@16.
   aForm ← aForm bitEdit.
   ```

4. Select these statements with the right mouse button and then select a "do it" from the middle button menu to execute the statements.
5. Use the Bit Editor to create the cursor form you want.

6. Use the middle mouse button to "accept" the form when you are done with it.
7. Enter this following smalltalk statement in the workspace:
   ```
   aForm writeOn: 'newcursor.form'.
   ```
8. Select the statement and then "do it". This should create the file newcursor.form.
9. Now exit smalltalk with the middle button menu.

You can use the graphics library to read newcursor.form and use the form as a new cursor form. The following C program excerpt reads the new cursor from a file and installs it as the graphics cursor.

```
struct FORM *cursor;
cursor = ReadForm("newcursor.form");
SetCursor(cursor);
```

## System Fonts

### Font Styles and Layout

The 4405 and 4406 AIM systems support two types of fonts in the directory */fonts*:

- Proportional fonts, in which character cells vary in height and width for each character.
- Monospaced fonts, in which character cells are the same height and width for each character.

The proportional fonts come in two faces (Serif and Sans-Serif) and four styles (bold, italic, bold italic, and regular) in a variety of point sizes (8, 10, 12, 14, 18, 24, and 36). Each proportional font is either Pellucida[1] Serif or Pellucida Sans-Serif. The character set for any proportional font is listed in Figure 4-4, *Tektronix Proportional Fonts (PellucidaSerif and PellucidaSans-Serif).*

The monospaced fonts are named Pellucida Typewriter and come in four sizes (10, 12, 16, and 18 point) and two styles (bold and regular). The character set for Pellucida Typewriter is listed in Figure 4-5, *Tektronix Monospaced Fonts (Pellucida Typewriter) Part 1* and Figure 4-6, *Tektronix Monospaced Fonts (Pellucida Typewriter) Part 2.*

A few notes on interpreting the font tables will be helpful in contructing an application. The spaces in the table that are blank do not have a printing character for the corresponding character code. The characters for ASCII 32 through ASCII 127 are present in bother the monospaced and proportional fonts. The proportional fonts contain additional characters in ASCII 1 through ASCII 31. Many of these characters are compatible with those originally supplied by Xerox in the original Smalltalk image.

1. Pellucida is a registered trademark of Bigelow and Holmes

"m space" is a blank character which is the height and width of the letter m. "n space" is a blank character which is the height and width of the letter n. "em" and "en" are dashes the width of the character "m" and "n", respectively.

Smalltalk StrikeFont class has methods for reading and writing Tektronix font files. Note that whenever Smalltalk reads a Tektronix font file, it switches the character position of the uparrow character (↑) and left arrow (←) with the caret (^) and underscore (_) characters. Thus, if you ask, for instance, the character ↑ what its asciiValue is, you get 94.

The method to write a StrikeFont takes care to switch the positions of the ↑, ←, ^, and _ characters if the type of the strike font is either 1 (Tektronix monospaced) or 2 (Tektronix proportionally spaced). This ensures that the proportional or monospaced fonts written by Smalltalk have consistent character ordering.

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|---|
| 0 | | ~ | space | 0 | @ | P | ‘ | p |
| 1 | ˇ | ffi | ! | 1 | A | Q | a | q |
| 2 | ¿ | ffl | ” | 2 | B | R | b | r |
| 3 | Ç | <u>em</u> | # | 3 | C | S | c | s |
| 4 | ¨ | fi | $ | 4 | D | T | d | t |
| 5 | ` | fl | % | 5 | E | U | e | u |
| 6 | ff | <u>en</u> | & | 6 | F | V | f | v |
| 7 | ‘ | ˘ | ’ | 7 | G | W | g | w |
| 8 | ¡ | — | ( | 8 | H | X | h | x |
| 9 | | n space | ) | 9 | I | Y | i | y |
| 10 | | ∞ | * | : | J | Z | j | z |
| 11 | ´ | ↑ | + | ; | K | [ | k | { |
| 12 | | ← | , | < | L | \ | \| | \| |
| 13 | | . | - | = | M | ] | m | } |
| 14 | ¯ | ~ | . | > | N | ^ | n | ~ |
| 15 | m space | ° | / | ? | O | _ | o | █ |

**Figure 4-4. Tektronix Proportional Fonts (PellucidaSerif and PellucidaSans-Serif).**

|  | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|---|
| 0 |  |  | space | 0 | @ | P | ‘ | p |
| 1 |  |  | ! | 1 | A | Q | a | q |
| 2 |  |  | ” | 2 | B | R | b | r |
| 3 |  |  | # | 3 | C | S | c | s |
| 4 |  |  | $ | 4 | D | T | d | t |
| 5 |  |  | % | 5 | E | U | e | u |
| 6 |  |  | & | 6 | F | V | f | v |
| 7 |  |  | ’ | 7 | G | W | g | w |
| 8 |  |  | ( | 8 | H | X | h | x |
| 9 |  | n space | ) | 9 | I | Y | i | y |
| 10 |  | ↑ | * | : | J | Z | j | z |
| 11 |  | ← | + | ; | K | [ | k | { |
| 12 |  |  | , | < | L | \ | \| | \| |
| 13 |  |  | - | = | M | ] | m | } |
| 14 |  |  | . | > | N | ^ | n | ~ |
| 15 | m space |  | / | ? | O | _ | o | ■ |

**Figure 4-5. Tektronix Monospaced Fonts (Pellucida Typewriter) Part 1.**

| B8 B7 B6 B5 BITS B4 | B3 | B2 | B1 | 1 0 0 0 | 1 0 0 1 | 1 0 1 0 | 1 0 1 1 | 1 1 0 0 | 1 1 0 1 | 1 1 1 0 | 1 1 1 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | NU 128 | DL 144 | SP 160 | 0 176 | – 192 | Ñ 208 | ◆ 224 | ⊟ 240 |
| 0 | 0 | 0 | 1 | SH 129 | D1 145 | Ä 161 | 1 177 | ¢ 193 | ñ 209 | █ 225 | ⊟ 241 |
| 0 | 0 | 1 | 0 | SX 130 | D2 146 | ä 162 | 2 178 | ¦ 194 | ¿ 210 | HT 226 | ⊟ 242 |
| 0 | 0 | 1 | 1 | EX 131 | D3 147 | Å 163 | 3 179 | † 195 | i 211 | FF 227 | □ 243 |
| 0 | 1 | 0 | 0 | ET 132 | DA 148 | å 164 | 4 180 | □ 196 | α 212 | CR 228 | ◫ 244 |
| 0 | 1 | 0 | 1 | EQ 133 | NK 149 | Æ 165 | 5 181 | ■ 197 | σ 213 | LF 229 | ◫ 245 |
| 0 | 1 | 1 | 0 | AK 134 | SY 150 | æ 166 | 6 182 | ● 198 | τ 214 | ° 230 | ⊞ 246 |
| 0 | 1 | 1 | 1 | BL 135 | EB 151 | à 167 | 7 183 | Δ 199 | ρ 215 | ± 231 | ⊞ 247 |
| 1 | 0 | 0 | 0 | BS 136 | CN 152 | Ç 168 | 8 184 | δ 200 | μ 216 | NL 232 | ◫ 248 |
| 1 | 0 | 0 | 1 | HT 137 | EM 153 | é 169 | 9 185 | λ 201 | Σ 217 | VT 233 | ≤ 249 |
| 1 | 0 | 1 | 0 | LF 138 | SB 154 | è 170 | ù 186 |  | Ω 218 | ⊡ 234 | ≥ 250 |
| 1 | 0 | 1 | 1 | VT 139 | EC 155 | Ö 171 | β 187 |  | ⌠ 219 | ⊡ 235 | π 251 |
| 1 | 1 | 0 | 0 | FF 140 | FS 156 | ö 172 | Θ 188 |  | ⌡ 220 | ⊡ 236 | ≠ 252 |
| 1 | 1 | 0 | 1 | CR 141 | GS 157 | ø 173 | ¤ 189 |  | ÷ 221 | ⊡ 237 | £ 253 |
| 1 | 1 | 1 | 0 | SO 142 | RS 158 | Ü 174 | § 190 | ¬ 206 | ≈ 222 | ⊞ 238 | ■ 254 |
| 1 | 1 | 1 | 1 | SI 143 | US 159 | ü 175 | ▪▪ 191 | ∝ 207 | ┌ 223 | ⊟ 239 | DT 255 |

Figure 4-6. Tektronix Monospaced Fonts (Pellucida Typewriter) Part 2.

## Graphical Text

Functions are provided which allow you to display text on the screen, or on any form, using any of the various fonts provided in /fonts. Simpler functions use the default font, extended versions of these functions allow you to specify a particular font.

Before a font is used it must be initialized, for example:

```
struct FontHeader *afont;
afont = FontOpen("/fonts/Pellucida12B.font");
```

This example loads the Pellucida Roman 12-point Bold font into afont. You can get a copy of the default font by passing FontInit a NULL argument.

```
afont = FontOpen(NULL);
```

When a font is no longer needed, use FontClose to release the associated storage.

You can use the function StringDraw to display text on the screen. The arguments to StringDraw are the string and the location at which to start displaying the string.

```
struct POINT pt;
pt.x = pt.y = 100;
StringDraw("Hello World",&pt);
```

A single character can be displayed using CharDraw. This function takes an ASCII character code and a pointer to a POINT structure as its arguments.

The extended versions of these functions, called StringDrawX and CharDrawX, require additional arguments. One additional argument is a pointer to a BBCOM structure which defines the destination form, destination rectangle (text bounding box), clipping rectangle and combination rule. Within the text bounding box, text is scrolled, if necessary, as it is displayed. Default behaviors for embedded tab and newline characters are also supported. The other additional argument specifies the font.

## Menus

### Pop-Up Menus

The graphics and events library provides two types of menus: text menus with string items and icon menus with form items. Menu creation functions – MenuCreate, MenuCreateX, IconMenuCreate, IconMenuCreateX – return a pointer to an initialized MENU structure. The selection function, MenuSelect, is passed a pointer to a MENU structure and returns the array index (zero-based) of the item selected. If no selection is made, then the number of menu items (one plus the last legal item index) is returned. A negative returned value signals an error. The MenuDestroy function takes a pointer to a MENU structure as its argument and releases the storage associated with a menu.

### Text Menus

The following C program creates a simple menu, displays it on the screen, waits for the user to select a menu item, and prints the returned value.

```
#include <graphics.h>
#include <font.h>

char *item[5] = {"zero","one","two","three","four"};

main()
{
   struct MENU *menu;
   short choice;

   menu = MenuCreate(5,menu_item);

   choice = MenuSelect(menu);
   printf("choice: %d0,choice);

   MenuDestroy(menu);
}
```

The next example uses the MenuCreateX function. This function requires additional arguments which specify menu options, the initial highlighted item, and the font for displaying menu items. Menu options are defined by an array of flags. Currently supported options include MENU_LINE (a line is drawn below the item) and MENU_NOSELECT – the item cannot be selected.

Menus remember the item selected the last time the menu was used; the mouse is positioned over this item.

```
#include <graphics.h>
#include <math.h>
#include <font.h>

char *items[6] = {"quit","line","rect","box","circle","clear"};
int flags[6] = {MENU_LINE,0,0,0,MENU_LINE,0};  /* menu includes two lines */

struct RECT rect1 = {50,100,500,180};
struct RECT rect2 = {250,50,100,200};
struct POINT pt1 = {400,240};
struct POINT pt2 = {0,200};
struct POINT pt3 = {640,200};

main()
{
   struct MENU *menu;
   struct FontHeader *font;
   short choice;

   font = FontOpen(NULL);   /* returns default font */
   menu = MenuCreateX(6,items,flags,0,font);
   InitGraphics(1);
   printf("press any button for menu0);

   while (TRUE) {  /* loop for menu options */
      if (GetButtons()) {
         choice = MenuSelect(menu);
         if (choice == 0) break;  /* terminate if "quit" selected */
         else switch (choice) {
            case 1:  LineDraw(&pt2,&pt3); break;
            case 2:  RectDraw(&rect2); break;
            case 3:  RectBoxDraw(&rect1,3); break;
            case 4:  CircleDraw(&pt1,100); break;
            case 5:  ClearScreen(); break;
            }
         }
   }
   ClearScreen();
   ExitGraphics();
   MenuDestroy(menu);
}
```

## Icon Menus

The following C program demonstrates the creation and use of an icon menu. In this program, a menu is created from the standard cursor forms. Each time the user selects from the menu, the cursor is changed. When the program terminates, the previous display state (including cursor) is restored.

```
#include <graphics.h>
#include <font.h>

main()
{
   struct MENU *menu;
   struct FORM *items[5], oldcursor;
   struct DISPSTATE ds;
   short choice;

   items[0] = &NormalCursor;  /* initialize menu items and menu */
   items[1] = &WaitCursor;
   items[2] = &CrosshairCursor;
   items[3] = &OriginCursor;
   items[4] = &CornerCursor;
   menu = IconMenuCreate(5,items);

   SaveDisplayState(&ds);  /* initialize display */
   InitGraphics(1);
   printf("press any button for menu0);
   printf("release button out of menu to exit0);

   while (TRUE) {  /* loop changing cursors */
         if (GetButtons()) {
            choice = MenuSelect(menu);
            if (choice == 5) break;  /* terminate if no item selected */
            else SetCursor(items[choice]);
         }
   }
   ClearScreen();  /* clean up display */
   ExitGraphics();
   RestoreDisplayState(&ds);
   MenuDestroy(menu);
}
```

## Event Processes

The graphics library includes support for event management. An event may be a key press, mouse button press, or time value. Smalltalk uses the event mechanism for management of all interactive input.

As an event is generated, it is inserted in the event queue. Functions are provided for turning event processing on and off, for accessing the next (oldest) event, and for returning the number of events in the queue. Figure 4-7, *Event Queue Processing*, shows how events are generated and processed.

It is possible to process mouse events only, leaving the keyboard generating ASCII key codes. The following C program excerpt turns on events, then sets the keyboard to ASCII, which leaves only mouse events to be placed in the queue.

```
EventEnable(1);
SetKBCode(1);
```

The function EGetNext() is used to return the next value in the event queue. Since some events require one value and some require three values, this is either a complete event, an event header, or half of a long time event parameter. The event type is an integer in the range -1 to 5, inclusive. A negative value signals an error. Here are the event type definitions from /lib/include/graphics.h:

```
#define E_DELTATIME          0
#define E_XMOUSE             1
#define E_YMOUSE             2
#define E_PRESS              3
#define E_RELEASE            4
#define E_ABSTIME            5
```

Type values 1 through 4 indicate that the event parameter is embedded in the event value. Type values 0 and 5 indicate that the parameter is given by the next two values in the event queue.

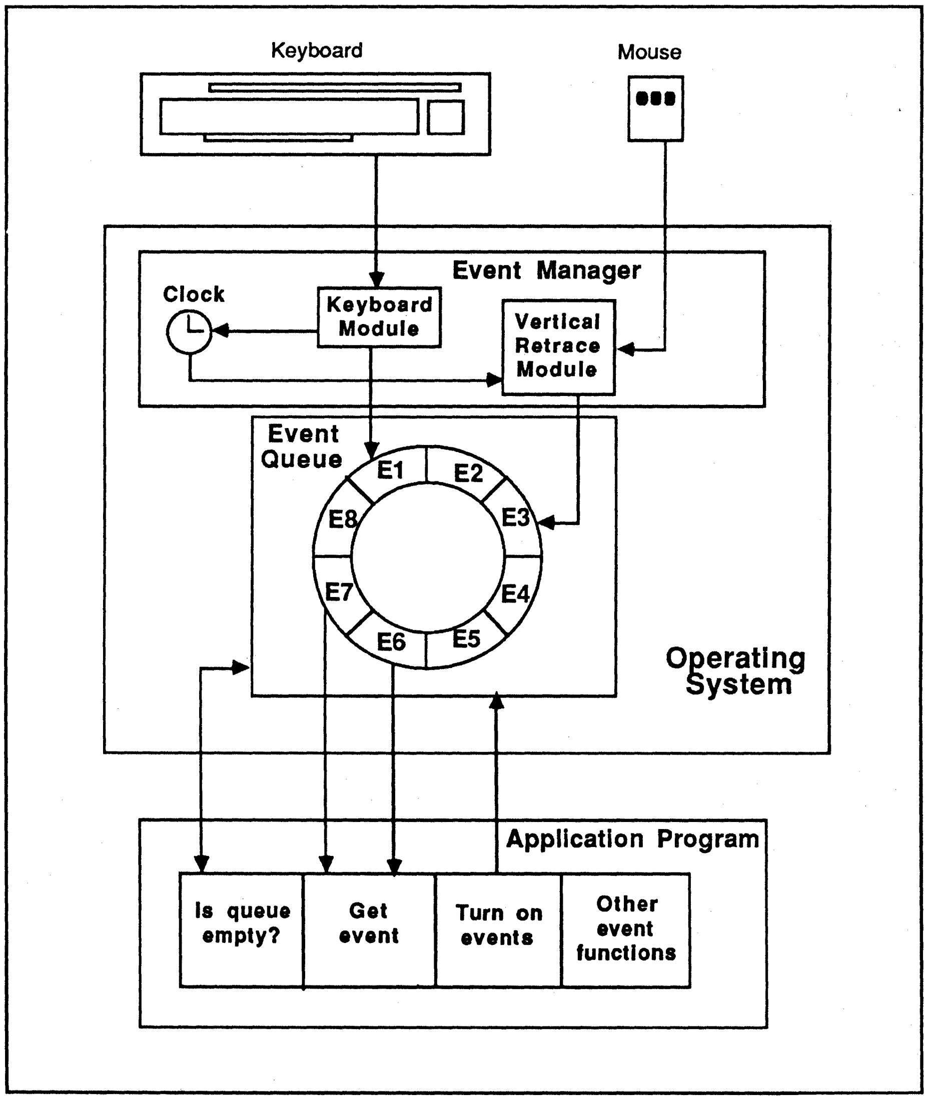


**Figure 4-7. Event Queue Processing.**

# Section 5
# Graphics Library Reference

## Graphics and Events Library

The graphics library provides access to the bit-mapped display and to the event manager. It uses the mechanisms added to the 4400 Series operating system which support Smalltalk´s use of the bit-mapped display the keyboard, and the mouse. The graphics library allows applications to use the "BitBlt" graphics primitive, change the cursor, detect button presses, and perform simple graphics operations such as draw lines and boxes, as well as other related abilities.

The library itself exists in the file named */lib/graphics*, with C header files which define the various structures in */lib/include/graphics.h* and */lib/include/font.h*.

In the description of the graphics library functions, the following conditions apply:

- All arguments are of the type *int*, unless otherwise specified.
- In all the following descriptions, the C language definitions of true and false are valid.
- For true/false, arguments are interpreted by the functions as true <> 0 and false == 0.
- The values returned from the library functions should be interpreted as true > 0, false == 0, and error condition < 0.
- All functions without explicit return values will return success/failure indications as success == 0, failure (error condition) < 0.
- Any function which returns an error condition will also set the global variable *errno* to an appropriate error code.

## About This Section

This section is made up of manual page descriptions for each graphics library function. The functions are alphabetically arranged. In addition to the manual pages for each function, you will find at the head of the manual pages an alphabetical list of all the functions. This list includes each function´s name, a symbolic listing of its arguments, and a short description of its operation. The symbolic arguments list shows the number and type of each argument. See Table 5-1, *Symbolic Arguments*, for the meanings of the symbolic arguments.

**Table 5-1**
*Symbolic Arguments*

| Symbol | Meaning |
|---|---|
| *b* | struct BBCOM *b; |
| *c* | char c; |
| *d* | struct DISPSTATE *d; |
| *f* | struct FORM *f; |
| *fa* | struct FORM **fa; (array of forms) |
| *fh* | struct FontHeader *f; |
| *i* | int i; |
| *ip* | int *ip; (pointer to int) |
| *l* | unsigned long l; |
| *m* | struct MENU *m; |
| *p* | struct POINT *p; |
| *pa* | struct POINT *pa; (array of points) |
| *q* | struct QUADRECT *q; |
| *r* | struct RECT *r; |
| *s* | char *s; (string) |
| *sa* | char **sa; (array of strings) |

## List of Functions

| | |
|---|---|
| **BbcomDefault(*b*)** | Initializes a *BBCOM* structure. |
| **BbcomPrint(*b*)** | Prints values in a *BBCOM* structure. |
| **BitBlt(*b*)** | Performs the BitBlt operation with a specified *BBCOM* structure. |
| **CharDraw(*c,p*)** | Draws a character on the display. |
| **CharDrawRawX(*c,p,b,fh*)** | Draws a character using the specified arguments exclusive of the destination rectangle. |
| **CharDrawX(*c,p,b,fh*)** | Draws a character using the specified arguments including the destination rectangle (text bounding box). |
| **CharWidth(*c,fh*)** | Returns the width in pixels required to draw a character with a specified font. |
| **CircleDraw(*p,i*)** | Draws a circle on the display. |
| **CircleDrawX(*p,i,i,b*)** | Draws a circle using the specified *BBCOM* structure. |
| **ClearScreen()** | Blanks( or "clears") the display. |
| **CursorTrack(*i*)** | Makes the cursor track with the mouse. |
| **CursorVisible(*i*)** | Makes the cursor visible. |

| | |
|---|---|
| **DisplayVisible(*i*)** | Makes the display visible. |
| **EClearAlarm()** | Clears any pending alarms that a process has requested. |
| **EGetCount()** | Returns event values from the event buffer. |
| **EGetNewCount()** | Returns event values from the event buffer since the previous call to **EGetNewCount**. |
| **EGetNext()** | Returns the next value in the event buffer. |
| **EGetTime()** | Returns the system time. |
| **ESetAlarm(*i*)** | Requests a signal after the specified time has elapsed. |
| **ESetSignal()** | Requests signals when events occur. |
| **EventDisable()** | Disables event processing. |
| **EventEnable()** | Enables event processing. |
| **ExitGraphics()** | Terminates use of graphics mode. |
| **FontClose(*fh*)** | Releases font storage memory to the system. |
| **FontOpen(*s*)** | Initializes a font from a font file. |
| **FormCopy(*f,f*)** | Copies one form into another form. |
| **FormCreate(*i,i*)** | Allocates and initializes a *FORM* structure and bitmap. |
| **FormDestroy(*f*)** | Deallocates a *FORM* structure and bitmap. |
| **FormFromUser()** | Returns a form from the display. |
| **FormGetPoint(*f,p*)** | Returns the value of a specified pixel in a form. |
| **FormPrint(*f*)** | Prints the values in a specified *FORM* structure. |
| **FormRead(*s*)** | Reads a specified file and returns the form in it. |
| **FormSetPoint(*f,p,i*)** | Sets the value of a specified pixel in a form. |
| **FormWrite(*s*)** | Writes the specified form to a file. |
| **GetButtons()** | Returns a value encoding the current state of the mouse buttons. |
| **GetCPosition(*p*)** | Returns the current cursor position in a point. |
| **GetCursor(*f*)** | Copies the current cursor information to a form. |
| **GetMachineType()** | Returns the 4400 Series model number as it is set at machine initialization time or by **SetMachineType**. |
| **GetMBounds(*p,p*)** | Returns the current mouse bounds. |
| **GetMPosition(*p*)** | Returns the current mouse position. |
| **GetRealMachineType()** | Returns the 4400 Series model number as set at machine initialization time. |
| **GetTermEmRC(*ip,ip*)** | Returns the current size of the terminal emulator. |

| | |
|---|---|
| GetViewport(*p*) | Returns in a point the upper left corner of the viewport. |
| IconMenuCreate(*i,fa*) | Initializes a *MENU* structure with items being *FORM* structures. |
| IconMenuCreateX(*i,fa,ip,i*) | Initializes a *MENU* structure with arguments additional to **IconMenuCreate.** |
| InitGraphics(*i*) | Maps the bit-mapped display into the calling process's space and otherwise initializes graphics mode. |
| LineDraw(*p,p*) | Draws a line from one specified point to another. |
| LineDrawX(*p,p,i,i,b*) | Draws a line from one specified point to another with variable width and a specified *BBCOM* structure. |
| MenuCreate(*i,sa*) | Initializes a *MENU* structure with items being strings. |
| MenuCreateX(*i,sa,ip,i,fh*) | Initializes a *MENU* structure with arguments additional to **MenuCreate.** |
| MenuDestroy(*m*) | Releases storage used for a *MENU* structure. |
| MenuPrint(*m*) | Prints the values in the fields of a *MENU* structure. |
| MenuSelect(*m*) | Pops up the menu and allows the user to select an option. |
| PaintLine(*b,p*) | Draws a line on the display. See **LineDraw.** |
| PanCursorEnable(*i*) | Turns cursor panning on or off. |
| PanDiskEnable(*i*) | Turns joydisk panning on or off. |
| PointDistance(*p,p*) | Returns the distance between two points. |
| PointFromUser(*p*) | Returns a display location selected by the user with a mouse click. |
| PointMax(*p,p,p*) | Returns the lower right corner of a rectangle defined by two points. |
| PointMidpoint(*p,p,p*) | Returns the midpoint of a line. |
| PointMin(*p,p,p*) | Returns the upper left corner of a rectangle defined by two points. |
| PointPrint(*p*) | Prints the address and values of a point. |
| PointsToRect(*p,p,r*) | Returns the smallest rectangle containing two specified points. |
| PointToRC(*ip,ip,p*) | Given a point on the display, returns the character cell that point is in. |
| PolygonDraw(*i,pa*) | Draws a filled-in polygon. |
| PolygonDrawX(*i,pa,b*) | Draws a filled-in polygon using the specified *BBCOM* structure. |
| PolyLineDraw(*i,pa*) | Draws an *un*filled-in polygon. |
| PolyLineDrawX(*i,pa,i,i,b*) | Draws an *un*filled-in polygon using the specified *BBCOM* structure. |

| | |
|---|---|
| **ProtectCursor**(*r*,*r*) | Removes the cursor from either of two specified display rectangles. |
| **QuadrectPrint**(*q*) | Prints the address and values in the specified *QUADRECT* structure. |
| **RCToRect**(*r*,*ip*,*ip*) | Converts row and column indices to a rectangle defining a character cell. |
| **RectAreasDiffering**(*r*,*r*,*q*) | Returns the non-intersecting portions of two specified rectangles. |
| **RectAreasOutside**(*r*,*r*,*q*) | Returns the non-intersecting portion of one of two specified rectangles. |
| **RectBoxDraw**(*r*,*i*) | Draws a box around a specified rectangle. |
| **RectBoxDrawX**(*r*,*i*,*b*) | Draws a box around a specified rectangle with the specified *BBCOM* structure. |
| **RectContainsPoint**(*r*,*p*) | Returns true if a rectangle contains the specified point. |
| **RectContainsRect**(*r*,*r*) | Returns true if a rectangle contains another specified rectangle. |
| **RectDraw**(*r*) | Draws a solid rectangle. |
| **RectDrawX**(*r*,*b*) | Draws a solid rectangle with the specified *BBCOM* structure. |
| **RectFromUser**(*r*) | Returns a rectangular region specified on the display by a user. |
| **RectFromUserX**(*p*,*f*,*r*) | Returns a rectangular region specified by a user with arguments additional to **RectFromUser**. |
| **RectIntersect**(*r*,*r*,*r*) | Returns the intersection rectangle of two specified rectangles. |
| **RectIntersects**(*r*,*r*) | Returns true if two rectangles intersect. |
| **RectMerge**(*r*,*r*,*r*) | Returns the smallest rectangle that can be drawn around two rectangles. |
| **RectPrint**(*r*) | Prints the address and values in a *RECT* structure. |
| **ReleaseCursor**() | Restores a cursor protected by a call to **ProtectCursor**. |
| **RestoreDisplayState**(*d*) | Restores the state of the display saved by a call to **SaveDisplayState**. |
| **SaveDisplayState**(*d*) | Stores the various display state variables in a *DISPSTATE* structure. |
| **ScreenSaverEnable**(*i*) | Sets the mode to blank or not blank the display after ten minutes of no keyboard activity. |
| **SetCPosition**(*p*) | Sets the cursor position to a specified point. |
| **SetCursor**(*f*) | Installs a new cursor. |

| | |
|---|---|
| SetKBCode(*i*) | Selects either ANSI (ASCII character) mode or event processing mode for keyboard activity reports. |
| SetMachineType(*i*) | Sets the value of the 4400 Series model number. |
| SetMBounds(*p*,*p*) | Sets the boundary of the region that contains the cursor. |
| SetMPosition(*p*) | Sets the mouse location. |
| SetViewport(*p*) | Specifies where the upper left corner of the viewport is to be. |
| StringDraw(*s*,*p*) | Draws a string on the display. |
| StringDrawRawX(*s*,*p*,*b*,*fh*) | Draws a string on the display without specifying a destination rectangle. |
| StringDrawX(*s*,*p*,*b*,*fh*) | Draws a string on the display with the destination rectangle (text bounding box) and other arguments specified. |
| StringWidth(*s*,*fh*) | Computes and returns the width in pixel required to draw string with a specified font. |
| TerminalEnable(*i*) | Enables or disables the terminal emulator. |
| VideoNormal(*i*) | Sets the display to normal (black on white) or reverse video. |

# BbcomDefault

## SYNOPSIS

```
#include <graphics.h>
BbcomDefault(bbcom)
     struct BBCOM *bbcom;
```

## Arguments

| | |
|---|---|
| <bbcom> | A pointer to a *BBCOM* structure. |

## Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Initializes a *BBCOM* structure to the following default values:

| | |
|---|---|
| source form | *NULL* |
| destination form | Screen |
| source point | *0,0* |
| destination rectangle | *0,0,ScrWidth,ScrHeight* |
| clipping rectangle | *0,0,ScrWidth,ScrHeight* |
| halftone form | *NULL* |
| combination rule | *bbSorD* |

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## BbcomPrint

### SYNOPSIS

```
#include <graphics.h>
BbcomPrint(bbcom)
      struct BBCOM *bbcom;
```

### Arguments

| | |
|---|---|
| <bbcom> | A pointer to a *BBCOM* structure. |

### Returns

Nothing is returned.

### DESCRIPTION

Prints the address (in hexadecimal) and the values of the fields in decimal of the structure pointed to by the *bbcom* argument.

### ERRORS REPORTED

No errors are reported.

# BitBlt

## SYNOPSIS

```
#include <graphics.h>
BitBlt(bitbltComPtr)
     struct BBCOM *bitbltComPtr;
```

### Arguments

| | |
|---|---|
| <bitbltComPtr> | A pointer to a *BBCOM* structure. |

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Perform the BitBlt command described in the record pointed to by the parameter. The record contains the source and destination rectangles, clipping regions, halftone mask, and combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Invalid parameter in structure.

# CharDraw

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
CharDraw(ch,loc)
      char ch;
      struct POINT *loc;
```

### Arguments

| | |
|---|---|
| <char> | An ASCII code for a character. |
| <loc> | The location at which the character is to be displayed. |

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *ch* using the default font onto the screen starting at the point *loc*. The text bounding box (and clipping rectangle) is defined by the current viewport. The *bbS* combination rule is used. The value of *loc* is updated to reflect the end of the displayed character.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# CharDrawRawX

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
CharDrawRawX(ch,loc,bbcom,font)
      char ch;
      struct POINT *loc;
      struct BBCOM *bbcom;
      struct FontHeader *font;
```

## Arguments

| | |
|---|---|
| <char> | An ASCII code for a character. |
| <loc> | The location at which the character is to be displayed. |
| <bbcom> | A structure that should define the destination form, clipping rectangle, halftone form, and combination rule. |
| <font> | The font to use for displaying the character. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *ch*, using *font*, onto a form starting at the point *loc*. The *bbcom* argument should include the destination form, clipping rectangle, halftoneform (optional) and combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## SEE ALSO

**CharDrawX**

# CharDrawX

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
CharDrawX(ch,loc,bbcom,font)
     char ch;
     struct POINT *loc;
     struct BBCOM *bbcom;
     struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <char> | An ASCII code for a character. |
| <loc> | The location at which the character is to be displayed. |
| <bbcom> | A structure that should define the destination form, destination rectangle, clipping rectangle, halftone form, and combination rule. |
| <font> | The font to use for displaying the character. |

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *ch*, using *font*, onto a form starting at the point *loc*. The *bbcom* argument should include the destination form, destination rectangle (text bounding box), clipping rectangle, halftoneform (optional) and combination rule. The value of *loc* is updated to reflect the end of the displayed character.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## SEE ALSO

**CharDrawRawX**

# CharWidth

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
int CharWidth(ch,font)
     char ch;
     struct FontHeader *font;
```

## Arguments

| | |
|---|---|
| <ch> | The ASCII code for a character. |
| <font> | The font to be used for accessing the character width. |

## Returns

Returns zero or positive if successful; otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero or positive, this value is the number of pixels required to display the specified character.

## DESCRIPTION

Returns the width in pixels required to draw *ch* with *font*.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# CircleDraw

## SYNOPSIS

```
#include <math.h>
#include <graphics.h>
CircleDraw(center,radius)
     struct POINT *center;
     int radius;
```

## Arguments

| | |
|---|---|
| <center> | The center point of the circle. |
| <radius> | The radius of the circle. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a circle defined by *center* and *radius* on the screen form using the *bbSorD* combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# CircleDrawX

## SYNOPSIS

```
#include <math.h>
#include <graphics.h>
CircleDrawX(center,radius,width,bb)
     struct POINT *center;
     int radius,width;
     struct BBCOM *bb;
```

## Arguments

| | |
|---|---|
| <center> | The center point of the circle. |
| <radius> | The radius of the circle. |
| <width> | The width for drawing the circle. |
| <bb> | A structure that should define the destination form, clipping rectangle, and combination rule. |

## Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a circle defined by *center* and *radius* onto a form. The circle is drawn with a line of *width* pixels. The *bb* argument should specify the destination form, clipping rectangle and combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## NOTES

When a value greater than one is specified for *width*, extra pixels are added below and to the right.

## ClearScreen

### SYNOPSIS

```
   #include <graphics.h>
   ClearScreen()
```

### Arguments

None.

### Returns

Returns zero if successful, otherwise -1 with *errno* set to the system or graphics error code.

### DESCRIPTION

Set the full screen bitmap to zeros. If the screen is set to normal video, this will result in a white screen. The terminal emulator is not affected by this call, i.e., the terminal emulator's idea of where to place its next character is unchanged.

### ERRORS REPORTED

Graphics not initialized.

### NOTES

Graphics must be previously initilized. **InitGraphics**

# CursorTrack

## SYNOPSIS

```
#include <graphics.h>
int CursorTrack(mode)
     int mode;
```

### Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or FALSE (zero). |

### Returns

Returns zero or positive if successful, otherwise −1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

If the mode argument is *TRUE*, the cursor is made to track the mouse. If *FALSE*, the cursor is totally independent of mouse motion. In either case, the previous setting is returned.

## ERRORS REPORTED

Display primitive failure.

# CursorVisible

## SYNOPSIS

```
#include <graphics.h>
int CursorVisible(mode)
     int mode;
```

## Arguments

|  |  |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

If the mode argument is *TRUE*, then the cursor is made visible. If *FALSE*, the cursor is made invisible. In both cases the previous state of the cursor is returned.

## ERRORS REPORTED

Display primitive failure.

## DisplayVisible

### SYNOPSIS

```
#include <graphics.h>
int DisplayVisible(mode)
     int mode;
```

### ARGUMENTS

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

### RETURNS

Returns zero or positive if successful, otherwise −1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

### DESCRIPTION

If the mode argument is *TRUE*, then the display is made visible. If *FALSE*, the display is blanked. In both cases the previous state of the display is returned.

### ERRORS REPORTED

Display primitive failure.

# EClearAlarm

## SYNOPSIS

```
#include <graphics.h>
EClearAlarm();
```

### Arguments

There are no arguments.

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Clears any pending alarms that the process has requested.

## ERRORS REPORTED

Event primitive failed.

# EGetCount

## SYNOPSIS

```
#include <graphics.h>
int EGetCount();
```

## Arguments

There are no arguments.

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the number of event values in the event buffer waiting to be processed. Return of a negative number indicates an error.

## ERRORS REPORTED

Event primitive failed.

# EGetNewCount

## SYNOPSIS

```
#include <graphics.h>
int EGetNewCount()
```

### Arguments

There are no arguments.

### Returns

Returns zero or positive if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the number of event values in the event buffer which have occurred since the previous call to this function.

## ERRORS REPORTED

Event primitive failed.

# EGetNext

## SYNOPSIS

```
#include <graphics.h>
union EVENTUNION EGetNext()
```

### Arguments

There are no arguments.

### Returns

Returns an *EVENTUNION* which is a complete event, an event header, or half of a long time event parameter.

## DESCRIPTION

**EGetNext** returns the next value in the event buffer. Since some events require one value and some require three values, this is either a complete event, an event header, or half of a long time event parameter. The event type is an integer in the range –1 to 5, inclusive. A negative value signals an error. Type values 1, 2, 3, and 4 indicate that the event parameter is embedded in the event value. Otherwise, (type 0 and 5) the embedded parameter field is ignored, and the parameter is represented by the next two values in the event buffer. The definition of the union *EVENTUNION* and the values for each event type are specified in */lib/include/graphics.h*.

## ERRORS REPORTED

Event primitive failed.

# EGetTime

## SYNOPSIS

```
#include <graphics.h>
unsigned long EGetTime()
```

### Arguments

There are no arguments.

### Returns

Returns zero or positive if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the time, in milliseconds, since the system was powered up. A return time of 0 indicates an error.

## ERRORS REPORTED

Event primitive failed.

# ESetAlarm

## SYNOPSIS

```
#include <graphics.h>
ESetAlarm(time)
     unsigned long time;
```

### Arguments

| | |
|---|---|
| <time> | A millisecond time value. |

### Returns

Nothing is returned.

## DESCRIPTION

Requests a signal when the specified time (relative to system turn on), in milliseconds, is reached.

## ERRORS REPORTED

Display primitive failure.

# ESetSignal

## SYNOPSIS

```
#include <graphics.h>
ESetSignal()
```

### Arguments

There are no arguments.

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Request the event manager to signal the current process when events occur. The event signal is disabled after being issued.

## ERRORS REPORTED

Event primitive failed.

# EventDisable

## SYNOPSIS

```
#include <graphics.h>
EventDisable()
```

### Arguments

There are no arguments.

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Disables event processing, i.e., turns off the event manager. Keyboard input through the "console" device and terminal emulator is re-enabled.

## ERRORS REPORTED

Event primitive failed.

# EventEnable

## SYNOPSIS

```
#include <graphics.h>
EventEnable()
```

### Arguments

There are no arguments.

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Enables event processing, i.e., turns on the event manager. Any subsequent user input action will cause event values to be created. Keyboard input through the "console" device and terminal emulator is disabled.

## ERRORS REPORTED

Event primitive failed.

# ExitGraphics

## SYNOPSIS

```
#include <graphics.h>
ExitGraphics()
```

### Arguments

There are no arguments.

### Returns

Nothing is returned.

## DESCRIPTION

Terminates use of graphics mode. The screen is mapped out of the user´s address space.

## ERRORS REPORTED

Display primitive failure.

# FontClose

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
FontClose(font)
     struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <font> | A structure used for font information. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Releases the storage used for the storage of the specified font.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# FontOpen

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
struct FontHeader *
FontOpen(filename)
     char *filename;
```

## Arguments

| | |
|---|---|
| <filename> | A string designating the name (and location) of a file defining a font. |

## Returns

Returns a pointer to a FontHeader structure if successful; otherwise, *NULL* with *errno* set to the system of graphics error code.

## DESCRIPTION

Initializes a font from a font file. Returns a pointer to the font header.

## ERRORS REPORTED

Invalid file type or file I/O error.

Invalid (*NULL*) structure pointer.

# FormCopy

## SYNOPSIS

```
#include <graphics.h>
FormCopy(form1,form2)
     struct FORM *form1,*form2;
```

### Arguments

| | |
|---|---|
| <form1> | The source form. |
| <form2> | The destination form. |

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system of graphics error code.

## DESCRIPTION

Copies the bitmap and offsets from *form1* to *form2*. Both forms must have the same height and width.

## ERRORS REPORTED

Invalid (*NULL*) structure parameter.

Invalid parameter in structure.

# FormCreate

## SYNOPSIS

```
#include <graphics.h>
struct FORM *
FormCreate(width,height)
     short int width,height;
```

### Arguments

| | |
|---|---|
| <width> | A short integer specifying the width in bits of the form bitmap. |
| <height> | A short integer specifying the height in bits of the form bitmap. |

### Returns

This function returns a pointer to the allocated and filled-in form structure. *NULL* (zero) is returned if the memory allocation fails or if width or height are not positive.

## DESCRIPTION

Given width and height, allocates and creates a struct FORM with the correct values in it. Width and height are specified in number of bits.

## ERRORS REPORTED

Memory allocation failure.

# FormDestroy

## SYNOPSIS

```
#include <graphics.h>
FormDestroy(form)
     struct FORM *form;
```

## Arguments

| | |
|---|---|
| <form> | A pointer to a form structure previously created by FormCreate. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Deletes and deallocates a form and its associated bitmap.

## ERRORS REPORTED

Invalid (*NULL*) structure argument.

## NOTES

The screen form, returned by the **InitGraphics** call, cannot be destroyed.

# FormFromUser

## SYNOPSIS

```
#include <graphics.h>
struct FORM *
FormFromUser()
```

## Arguments

## Returns

Returns a form copied from a region of the screen if successful, otherwise *NULL* with *errno* set to the system or graphics error code.

## DESCRIPTION

A *FORM* is returned which is a copy of a region selected from the screen by the user. **RectFromUserX** is called with *BlackMask* so that the selected region is inverted during the selection process. The maximum region selectable is determined by the current mouse bounds.

## ERRORS REPORTED

Memory allocation failure.

## SEE ALSO

**RectFromUserX**

# FormGetPoint

## SYNOPSIS

```
#include <graphics.h>
int FormGetPoint(form,point)
      struct FORM *form;
      struct POINT *point
```

### Arguments

| | |
|---|---|
| <form> | A pointer to a *FORM* structure. |
| <point> | An x,y pair designating one pixel in the form. |

### Returns

The value of the specified pixel (zero or one) if successful; otherwise, –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the value of a particular pixel in a form.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# FormPrint

## SYNOPSIS

```
#include <graphics.h>
FormPrint(form)
     struct FORM *form;
```

## Arguments

| | |
|---|---|
| <form> | A pointer to a *FORM* structure. |

## Returns

Nothing is returned.

## DESCRIPTION

Prints the address (in hexadecimal) and the values of the fields (in decimal) of the structure pointed to by the *form* argument.

## ERRORS REPORTED

No errors are reported.

## FormRead

### SYNOPSIS

```
#include <graphics.h>
struct FORM *
FormRead(filename)
     char *filename;
```

### Arguments

| | |
|---|---|
| <filename> | A string designating the name (and location) of a file defining a form. |

### Returns

Returns a pointer to a *FORM* structure if successful, otherwise *NULL* with *errno* set to the system or graphics error code.

### DESCRIPTION

If successful, a new form is read from the specified file and returned. Smalltalk form file format is expected, which is a sequence of 16 bit values followed by the form. The initial values are 1 (indicating a form file), width, height, offset width and offset height. These values are followed by the rows of the form, from top to bottom.

### ERRORS REPORTED

Memory allocation failure.

File type mismatch or file I/O error.

# FormSetPoint

## SYNOPSIS

```
#include <graphics.h>
FormSetPoint(form,point,value)
      struct FORM *form;
      struct POINT *point;
      int value;
```

### Arguments

| | |
|---|---|
| <form> | A pointer to a *FORM* structure. |
| <point> | An x,y position in the form. |
| <value> | A value (zero or one) to set the specified pixel. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Set the value of one pixel in a form.

## ERRORS REPORTED

Invalide (*NULL*) structure pointer.

# FormWrite

## SYNOPSIS

```
#include <graphics.h>
FormWrite(form,filename)
     struct FORM *form;
     char *filename;
```

### Arguments

| | |
|---|---|
| <form> | A pointer to a *FORM* structure. |
| <filename> | A string designating the name (and location) of the output file. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

The specified form is written to the specified file. Smalltalk form file format is used. (See **FormRead.**)

## ERRORS REPORTED

Invalid file type or file I/O error.

Invalid (*NULL*) structure pointer.

## SEE ALSO

**FormRead**

# GetButtons

## SYNOPSIS

```
W#include <graphics.h>
    int GetButtons()
```

## Arguments

None.

## Returns

If the value returned is zero or positive, it encodes the current state of the mouse buttons. Otherwise –1 is returned with *errno* set to the system or graphics error code.

## DESCRIPTION

The returned value encodes the state of the mouse buttons with the low three bits corresponding to the three mouse buttons. Bit 2 corresponds to the left button, bit 1 to the middle, and bit 0 (the lsb) to the right button. If the value of the bit is a one, then the mouse button is depressed, if a zero it is released.

## ERRORS REPORTED

Display primitive failed.

## GetCPosition

Gets location of cursor.

### SYNOPSIS

```
#include <graphics.h>
GetCPosition(point)
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <point> | A pointer to a POINT structure to be filled in with the current cursor location. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

### DESCRIPTION

Sets the *x* and *y* fields of *point* to the current cursor location.

### ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Display primitive failure.

### NOTES

If *CursorTrack* mode is enabled, this is the same as **GetMPosition.**

Cursor location ("hot spot") is defined by the position (upper left corner) where the cursor is displayed, plus the offsets defined in the cursor form.

# GetCursor

## SYNOPSIS

```
#include <graphics.h>
GetCursor(curp)
     struct FORM *curp;
```

## Arguments

| | |
|---|---|
| <curp> | A pointer to a *FORM* structure which must define a 16x16 array of bits which will be filled with the current cursor. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

The bitmap and offsets of the current cursor are copied into the form pointed to by *curp*.

## ERRORS REPORTED

Invalid (*NULL*) structure argument.

Invalid parameter in structure.

Display primitive failure.

## GetMachineType

### SYNOPSIS

```
#include <graphics.h>
int GetMachineType();
```

### Arguments

There are no arguments.

### Returns

Returns a hexadecimal value of the form 0x440x0000 if successful, otherwise –1 with *errno* set to the system or graphics error code.

### DESCRIPTION

Returns the 4400 Series model number as set at machine initialization or by **SetMachineType**. The lower word of the returned value is reserved for future use.

### ERRORS REPORTED

Display primitive failure.

### SEE ALSO

**GetRealMachineType, SetMachineType**

# GetMBounds

## SYNOPSIS

```
#include <graphics.h>
GetMBounds(ulpoint,lrpoint)
     struct POINT *ulpoint,*lrpoint;
```

## Arguments

| | |
|---|---|
| <ulpoint> | A pointer to a POINT structure in which the current x and y coordinates of the upper left corner of the mouse motion bounding box will be placed. |
| <lrpoint> | A pointer to a POINT structure in which the current x and y coordinates of the lower right corner of the mouse motion bounding box will be placed. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Gets current mouse bounds.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Display primitive failure.

# GetMPosition

## SYNOPSIS

```
#include <graphics.h>
GetMPosition(point)
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <point> | A pointer to a POINT structure to be filled in with the current x and y screen coordinates of the mouse position. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Gets current mouse locations.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Display primitive failure.

## NOTES

If *CursorTrack* mode is enabled, this is the same as **GetCPosition**.

Cursor location ("hot spot") is defined by the position (upper left corner) where the cursor is displayed, plus the offsets defined in the cursor form.

# GetRealMachineType

## SYNOPSIS

```
#include <graphics.h>
int GetRealMachineType();
```

## Arguments

There are no arguments.

## Returns

Returns a hexadecimal value of the form 0x440x0000 if successful, otherwise -1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the actual 4400 Series model number determined by consulting the internal ROM memory. The lower word of the returned value is reserved for future use.

## ERRORS REPORTED

Display primitive failure.

## SEE ALSO

**GetMachineType, SetMachineType**

# GetTermEmRC

## SYNOPSIS

```
#include <graphics.h>
GetTermEmRC(row, col)
     int *row, *col;
```

## Arguments

| | |
|---|---|
| <row> | A pointer to an integer to receive the number of rows. |
| <col> | A pointer to an integer to receive the number of columns. |

## Returns

Returns the current size (number of rows and columns) of the terminal emulator.

## DESCRIPTION

Returns the current size of the terminal emulator.

## ERRORS REPORTED

Display primitive failure.

# GetViewport

## SYNOPSIS

```
#include <graphics.h>
GetViewport(point)
     struct POINT *point;
```

## Arguments

| | |
|---|---|
| <point> | A pointer to a POINT structure to be filled in with the current x and y screen coordinates of the upper left corner of the viewport. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Sets the *x* and *y* fields of *point* to the coordinates of the point of the virtual screen which is currently displayed in the upper left corner of the viewport.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Display primitive failure.

# IconMenuCreate

## SYNOPSIS

```
#include <graphics.h>
struct MENU *
IconMenuCreate(count,item)
      int count;
      struct FORM **item;
```

## Arguments

| | |
|---|---|
| <count> | The number of menu items, an integer. |
| <item> | An array of pointers to *FORM* structures. |

## Returns

Returns the address of a MENU structure if successful, otherwise *NULL* with *errno* set to the system or graphics error code.

## DESCRIPTION

Initializes a MENU structure and returns a pointer to it. The *count* argument is the number of items in the menu. The *item* argument is an array of pointers to *FORM* structures that define the actual menu items. A line is drawn between each pair of menu items.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Memory allocation failure.

# IconMenuCreateX

## SYNOPSIS

```
#include <graphics.h>
struct MENU *IconMenuCreateX(count,items,flags,previtem)
     int count, *flags, previtem;
     struct FORM **items;
```

## Arguments

| | |
|---|---|
| <count> | The number of items in the menu. |
| <items> | An array of pointers to *FORM* structures. |
| <flags> | An optional array of flags for each item. |
| <previtem> | The item where the mouse should be positioned initially. |

## Returns

The address of a MENU structure if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Initializes and returns a pointer to a *MENU* structure. The *count* argument is the number of items in the menu. The *items* argument is an array of pointers to FORM structures which define the actual menu items. The positioning of each item is determined by the size of the item and the values of its offsets.

The *flags* array is used for specifying information about menu items. Flags which may be specified include *MENU_LINE* (draw a line after the item), *MENU_NOSELECT* (the menu item is not selectable), *MENU_LEFT* (left-justify the item), and *MENU_RIGHT* (right-justify the item). Items are centered by default.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Memory allocation failure.

## SEE ALSO

**MenuSelect, MenuDestroy**

# InitGraphics

## SYNOPSIS

```
#include <graphics.h>
struct FORM *
InitGraphics(mode)
     int mode;
```

## Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (positive) or *FALSE* (zero). |

## Returns

Returns a pointer to the *FORM* structure which defines the screen bitmap. A *NULL* (zero) is returned if the initialization fails.

## DESCRIPTION

If the mode argument is *FALSE*, the bit-mapped display is mapped into the calling process's address space. If the argument is *TRUE*, this mapping is done and then the display is cleared, made visible and set to normal video (black on white) with both mouse and joydisk panning enabled and the cursor position tracking the mouse.

## ERRORS REPORTED

Graphics initialization failure.

# LineDraw

## SYNOPSIS

```
#include <graphics.h>
LineDraw(*point1,*point2)
     struct POINT *point1,*point2;
```

### Arguments

| | |
|---|---|
| <point1> | Starting point for line. |
| <point2> | Ending point for line. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a one-pixel-wide line from *point1* to *point2* on the screen form using the *bbSorD* combination rule. Both endpoints are drawn.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# LineDrawX

## SYNOPSIS

```
#include <graphics.h>
LineDrawX(point1,point2,width,drawlast,bbcom)
     struct POINT *point1,*point2;
     int width,drawlast;
     struct BBCOM *bbcom;
```

### Arguments

| | |
|---|---|
| <point1> | Starting point for the line. |
| <point2> | Ending point for the line. |
| <width> | The width in pixels for the line. |
| <drawlast> | Value of zero or one – determines if the last point is drawn. |
| <bbcom> | Defines the destination form, the clipping rectangle, and the combination rule. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to system or graphics error code.

## DESCRIPTION

Draw a line of *width* pixels from *point1* to *point2* onto a form. The *bbcom* argument should specify the destination form, clipping rectangle, and combination rule.

If *drawlast* is zero and *width* is one, then the final point (*point2*) is not drawn.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# MenuCreate

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
struct MENU *
MenuCreate(count,item)
      int count;
      char **item;
```

## Arguments

| | |
|---|---|
| <count> | The number of menu items. |
| <item> | An array of string items. |

## Returns

Returns zero if successful, otherwise *NULL* with *errno* set to the system or graphics error code.

## DESCRIPTION

Initializes a MENU structure and returns a pointer to it. The *count* argument is the number of items in the menu. The *item* argument is an array of strings that define the actual menu items. The default font is used to paint the items onto the menu form.

## ERRORS REPORTED

Invalid (*NULL*) array pointer.

# MenuCreateX

## SYNOPSIS

```
#include <font.h>
#include <graphics.h>
struct MENU *MenuCreateX(count,items,flags,previtem,font)
     int count, *flags, previtem;
     char **items;
     struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <count> | The number of items in the menu. |
| <items> | An array of pointers to strings. |
| <flags | An optional array of flags for each item. |
| <previtem> | The item where the mouse should be positioned initially. |
| <font> | The font to use for displaying menu items. |

### Returns

The address of a *MENU* structure if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Initializes and returns a pointer to a *MENU* structure. The *count* argument is the number of items in the menu. The *items* argument is an array of pointers to the strings which define the actual menu items.

The *flags* array is used for specifying information about menu items. Flags which may be specified include *MENU_LINE* (draw a line after the item), *MENU_NOSELECT* (the menu item is not selectable), *MENU_LEFT* (left-justify the item), and *MENU_RIGHT* (right-justify the item). Items are centered by default.

## ERRORS REPORTED

Invalid (*NULL*) array pointer.

## SEE ALSO

MenuSelect, MenuDestroy

# MenuDestroy

## SYNOPSIS

```
#include <graphics.h>
MenuDestroy(menu)
     struct MENU *menu;
```

## Arguments

| | |
|---|---|
| <menu> | A pointer to a *MENU* structure. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Releases the storage used for a MENU structure. (This storage is allocated by **MenuCreate, MenuCreateX, IconMenuCreate,** or **IconMenuCreateX.**)

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# MenuPrint

## SYNOPSIS

```
#include <graphics.h>
MenuPrint(menu)
        struct MENU *menu;
```

## Arguments

| | |
|---|---|
| <menu> | A pointer to a *MENU* structure. |

## Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Prints the address (in hexadecimal) and the values of the fields (in decimal) of the structure pointed to by the *menu* argument.

## ERRORS REPORTED

Event primitive failed.

# MenuSelect

## SYNOPSIS

```
#include <graphics.h>
MenuSelect(menu)
      struct MENU *menu;
```

## Arguments

| | |
|---|---|
| <menu> | A pointer to a MENU structure. |

## Returns

Returns the array index (zero or positive) of the item selected or the number of items in the menu if no selection was made. If an error has occurred, then –1 is returned, and *errno* is set to the system or graphics error code.

## DESCRIPTION

Pops up the specified *menu* and waits for click (or release) of any mouse button. Returns the number of the item selected, or the number of item in the menu for no selection. The menu is displayed at the current mouse location, possibly adjusted to be within the current mouse bounds.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PaintLine

## SYNOPSIS

```
#include <graphics.h>
PaintLine(bbcom,point)
     struct BBCOM *bbcom;
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <bbcom> | A pointer to a *BBCOM* structure that defines the source form, destination form, the clipping rectangle, the combination rule, etc. |
| <point> | An x,y pair specifying the beginning point of the line drawn. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Paint a line on the display. A sequence of BitBlt operations is performed while stepping a pixel at a time from *point* to the point specified by the x and y values of the *bbcom* destination rectangle. The width and height values of the *bbcom* destination rectangle determine the size of the "brush" used for drawing the line.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## NOTES

If the one of the exclusive OR rules is specified, and the source is *NULL* (ones), the response will instead be as if the line was drawn by the above stepping method to a hidden bitmap and then that hidden bitmap was combined with the destination bitmap according to the specified rule.

# PanCursorEnable

## SYNOPSIS

```
#include <graphics.h>
int PanCursorEnable(mode)
     int mode;
```

## Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was *TRUE* or enabled.

## DESCRIPTION

If the mode argument is *TRUE*, then the cursor is set to pan the viewport when it runs into the edges. If the argument is *FALSE*, the panning with the cursor is disabled. In either case, the previous setting is returned.

## ERRORS REPORTED

Display primitive failure.

## PanDiskEnable

### SYNOPSIS

```
#include <graphics.h>
int PanDiskEnable(mode)
     int mode;
```

### Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

### Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

### DESCRIPTION

If the mode argument is *TRUE*, then panning of the viewport when the joydisk is pressed is enabled. If the argument is *FALSE*, the panning with the joydisk is disabled. In either case, the previous setting is returned.

### ERRORS REPORTED

Display primitive failure.

# PointDistance

## SYNOPSIS

```
#include <math.h>
#include <graphics.h>
int PointDistance(point1,point2)
      struct POINT *point1;
      struct POINT *point2;
```

### Arguments

| | |
|---|---|
| <point1> | A pointer to the starting point of a line. |
| <point2> | A pointer to the ending point of a line. |

### Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns the distance between *point1* and *point2*.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PointFromUser

## SYNOPSIS

```
#include <graphics.h>
PointFromUser(point)
     struct POINT *point;
```

## Arguments

| | |
|---|---|
| <point> | A pointer to a *POINT* structure to be used for the result. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns a screen location selected by the user. The cursor is changed to *CrosshairCursor*. When any button is clicked (or released), the position of the cursor is copied into *point*. Selectable points are determined by the current mouse bounds. Cursor visibility and tracking are enabled for the operation, and the previous cursor, cursor visibility, and tracking modes are restored when the operation is complete.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## PointMax

### SYNOPSIS

```
#include <graphics.h>
PointMax(point1,point2,point3)
     struct POINT *point1,*point2,*point3;
```

### Arguments

| | |
|---|---|
| <point1> | A pointer to one of two points defining a rectangle. |
| <point2> | A pointer to the other of two points defining a rectangle. |
| <point3> | A pointer to a point (the lower right corner of the rectangle defined by *point1* and *point2*) to be used for the result. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

### DESCRIPTION

Returns in *point3* the lower right corner of the rectangle defined by *point1* and *point2*.

### ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PointMidpoint

## SYNOPSIS

```
#include <graphics.h>
PointMidpoint(point1,point2,point3)
     struct POINT *point1,*point2,*point3;
```

### Arguments

| | |
|---|---|
| <point1> | A pointer to one endpoint of a line. |
| <point2> | A pointer to the other endpoint of a line. |
| <point3> | A pointer to a point (the point half way between the two endpoints of a line) to be used for the result. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns in *point3* the midpoint of the line defined by *point1* and *point2*.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

## PointMin

### SYNOPSIS

```
#include <graphics.h>
PointMin(point1,point2,point3)
     struct POINT *point1,*point2,*point3;
```

### Arguments

| | |
|---|---|
| <point1> | A pointer to one of two points defining a rectangle. |
| <point2> | A pointer to the other of two points defining a rectangle. |
| <point3> | A pointer to a point (the upper left corner of the rectangle defined by *point1* and *point2*) to be used for the result. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

### DESCRIPTION

Returns in *point3* the upper left corner of the rectangle defined by *point1* and *point2*.

### ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PointPrint

## SYNOPSIS

```
#include <graphics.h>
PointPrint(point)
     struct POINT *point;
```

## Arguments

| | |
|---|---|
| <point> | A pointer to a *POINT* structure. |

## Returns

Nothing is returned.

## DESCRIPTION

Prints the address (in hexadecimal) and the values of the fields (in decimal) of the structure pointed to by the *point* argument.

## ERRORS REPORTED

No errors are reported.

# PointsToRect

## SYNOPSIS

```
#include <graphics.h>
PointsToRect(point1,point2,rect)
     struct POINT *point1,*point2;
     struct RECT *rect;
```

## Arguments

| | |
|---|---|
| <point1> | A pointer to one of two points that define a rectangle. |
| <point2> | A pointer to the other of two points that define a rectangle. |
| <rect> | A pointer to a RECT structure (the minimum-sized rectangle containing *point1* and *point2*) to be used for the result. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns in *rect* the minimum rectangle which contains both *point1* and *point2*.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PointToRC

## SYNOPSIS

```
#include <graphics.h>
PointToRC(row,col,point)
     int *row,*col;
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <row> | A pointer to an integer to receive the returned row index of the specified coordinate. |
| <col> | A pointer to an integer to receive the returned column index of the specified coordinate. |
| <point> | A pointer to a POINT structure which specifies an x and y screen coordinate. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Given a screen coordinate, returns the row and column indices of the character cell which that coordinate is in. (Row and column indices start with upper left of 1,1.)

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PolygonDraw

## SYNOPSIS

```
#include <math.h>
#include <graphics.h>
PolygonDraw(count,point)
     int count;
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <count> | The number of vertices (or edges) in the polygon. |
| <point> | An array of point structures. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws a filled-in polygon defined by the *point* array onto the screen using the *bbSorD* combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

# PolygonDrawX

## SYNOPSIS

```
#include <math.h>
#include <graphics.h>
PolygonDrawX(count,point,bbcom)
     int count;
     struct POINT *point;
     struct BBCOM *bbcom;
```

### Arguments

| | |
|---|---|
| <count> | The number of vertices (or edges) in the polygon. |
| <point> | An array of point structures. |
| <bbcom> | Should specify destination form, clipping rectangle halftone form and combination rule. |

### Returns

Returns zero if successful, otherwise −1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws a filled-in polygon defined by the *point* array. The *bbcom* argument should specify the destination form, clipping rectangle, halftoneform and combination rule.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Invalid parameter in structure.

# PolyLineDraw

## SYNOPSIS

```
#include <graphics.h>
PolyLineDraw(count,point)
        int count;
        struct POINT *point;
```

## Arguments

| | |
|---|---|
| <count> | The number of vertices (or edges) in the polygon. |
| <point> | An array of point structures. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Lines from *point[0]* to *point[1]* to ... *point[count-1]* to *point[0]* are drawn onto the screen using the *bbSorD* combination rule. Lines are drawn one pixel wide.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# PolyLineDrawX

## SYNOPSIS

```
#include <graphics.h>
PolyLineDrawX(count,point,width,closed,bbcom)
      int count,width,closed;
      struct POINT *point;
      struct BBCOM *bbcom;
```

## Arguments

| | |
|---|---|
| <count> | The number of vertices (or edges) in the polygon. |
| <point> | An array of point structures. |
| <width> | The width in pixels for drawing each edge. |
| <closed> | Determines if last point is connected to the first point. |
| <bbcom> | Should specify destination form, clipping rectangle halftone form and combination rule. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Lines from *point[0]* to *point[1]* to ... *point[count-1]* are drawn onto a form. Lines are drawn *width* pixels wide. If the value of *closed* is not 0, then an additional line is drawn from *point[count-1]* to *point[0]*. The last endpoint is not drawn. The *bbcom* argument should specify the destination form, clipping rectangle and combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Invalid parameter in structure.

# ProtectCursor

## SYNOPSIS

```
#include <graphics.h>
ProtectCursor(r1,r2)
     struct RECT *r1,*r2;
```

## Arguments

| | |
|---|---|
| <r1> | A pointer to a *RECT* structure. |
| <r2> | A pointer to a *RECT* structure. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Tell the operating system that graphics operations will be occurring in one or both of the screen areas defined by the two rectangles (either rectangle pointer may be null). The operating system will respond by removing the cursor from the screen if it is in either of the two areas. This instruction and its release (**ReleaseCursor**) should be used if the user is writing or reading directly from the screen. This cursor protection is already included in the routines of this library which draw on the screen.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# QuadrectPrint

## SYNOPSIS

```
#include <graphics.h>
QuadrectPrint(quadrect)
     struct QUADRECT *quadrect;
```

### Arguments

| | |
|---|---|
| <quadrect> | A pointer to a *QUADRECT* structure. |

### Returns

Nothing is returned.

## DESCRIPTION

The address (in hexadecimal) and the values of the fields (in decimal) of the structure pointed to by the *quadrect* argument are printed.

## ERRORS REPORTED

No errors are reported.

# RCToRect

## SYNOPSIS

```
#include <graphics.h>
RCToRect(rect,row,col)
      struct RECT *rect;
      int row,col;
```

## Arguments

| | |
|---|---|
| <rect> | A pointer to a RECT structure to receive the *x* and y screen coordinate of the upper left corner of the specified character cell and the width and height of that cell. |
| <row> | An integer specifying a row index of a character cell. |
| <col> | An integer specifying a column index of a character cell. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Convert row, column indices to a rectangle defining the character cell.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectAreasDiffering

## SYNOPSIS

```
#include <graphics.h>
RectAreasDiffering(rect1,rect2,quadrect)
     struct RECT *rect1,*rect2;
     struct QUADRECT *quadrect;
```

### Arguments

| | |
|---|---|
| <rect1>, <rect2> | Pointers to *RECT* structures. |
| <quadrect> | A pointer to a *QUADRECT* structure. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

The regions of *rect1* which are outside of *rect2*, and the regions of *rect2* which are outside of *rect1* are returned in *quadrect.*

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectAreasOutside

## SYNOPSIS

```
#include <graphics.h>
RectAreasOutside(rect1,rect2,quadrect)
     struct RECT *rect1,*rect2;
     struct QUADRECT *quadrect;
```

## Arguments

| | |
|---|---|
| <rect1>, <rect2> | Pointers to *RECT* structures. |
| <quadrect> | A pointer to a *QUADRECT* structure. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

The regions of *rect1* which are outside of *rect2* are returned in *quadrect.*

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectBoxDraw

## SYNOPSIS

```
#include <graphics.h>
RectBoxDraw(rect,width)
      struct RECT *rect;
      int width;
```

## Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |
| <width> | The width in pixels of the line used to draw the box. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a box *width* pixels wide on the screen form around *rect* using the *bbSorD* combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectBoxDrawX

## SYNOPSIS

```
#include <graphics.h>
RectBoxDrawX(rect,width,bbcom)
     struct RECT *rect;
     int width;
     struct BBCOM *bbcom;
```

### Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |
| <width> | The width in pixels of the line used to draw the box. |
| <bbcom> | Defines the destination form, the clipping rectangle, halftone form and the combination rule. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a box *width* pixels wide onto a form around *rect*. The *bb* argument should specify the destination form, clipping rectangle, halftone form, and combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectContainsPoint

## SYNOPSIS

```
#include <graphics.h>
int RectContainsPoint(rect,point)
     struct RECT *rect;
     struct POINT *point;
```

## Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |
| <point> | A pointer to a *point* structure. |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns 1 if *rect* contains *point*, otherwise 0.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectContainsRect

## SYNOPSIS

```
#include <graphics.h>
int RectContainsRect(rect1,rect2)
     struct RECT *rect1,*rect2;
```

### Arguments

| | |
|---|---|
| <rect1>, <rect2> | Pointers to *RECT* structures. |

### Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns 1 if *rect1* contains *rect2*, otherwise 0.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectDraw

## SYNOPSIS

```
#include <graphics.h>
RectDraw(rect)
     struct RECT *rect;
```

## Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a solid rectangle on the screen form using the *bbS* combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectDrawX

## SYNOPSIS

```
#include <graphics.h>
RectDrawX(rect,bbcom)
      struct RECT *rect;
      struct BBCOM *bbcom;
```

### Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |
| <bbcom> | Defines the distination form, clipping rectangle, halftone form, and combination rule. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draw a rectangle onto a form. The *bb* argument should specify the destination form, clipping rectangle, halftone form and combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectFromUser

## SYNOPSIS

```
#include <graphics.h>
RectFromUser(rect)
     struct RECT *rect;
```

### Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure to be used for the result. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

A region selected by the user is returned in the *rect* argument. The cursor is changed to *OriginCursor.* When the left mouse button is pressed, the upper left corner of the region is fixed, and the cursor is changed to *CornerCursor. GrayMask* is used as the halftoneform to indicate the selected region. Releasing the mouse button fixes the lower right corner of the region. The current mouse bounds define the maximum region that may be selected. Cursor visibility and tracking are enabled for the operation, and the previous cursor, cursor visibility, and tracking modes are restored when the operation is complete.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectFromUserX

## SYNOPSIS

```
#include <graphics.h>
RectFromUserX(minsize,mask,rect)
      struct POINT *minsize;
      struct FORM *mask;
      struct RECT *rect;
```

## Arguments

| | |
|---|---|
| <minsize> | The minimum height and width. |
| <mask> | The halftone form to use for highlighting the selected region. |
| <rect> | A pointer to a structure to use for the result. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

A region selected by the user is returned in the *rect* argument. The cursor is changed to *OriginCursor*. When the left mouse button is pressed, the upper left corner of the region is fixed, and the cursor is changed to *CornerCursor*. The *mask* argument, which should be a 16 by 16 bit form, is used as the halftoneform to indicate the selected region. After the upper left corner is fixed, the region is initialized to the size indicated with the *minsize* argument, and constrained to be no smaller. Releasing the mouse button fixes the lower right corner of the region. The current mouse bounds define the maximum region that may be selected. Cursor visibility and tracking are enabled for the operation, and the previous cursor, cursor visibility, and tracking modes are restored when the operation is complete.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectIntersect

## SYNOPSIS

```
#include <graphics.h>
RectIntersect(rect1,rect2,rect3)
     struct RECT *rect1,*rect2,*rect3;
```

## Arguments

| | |
|---|---|
| <rect1>, <rect2> | The two *RECT* structures to be compared. |
| <rect3> | A structure to hold the result. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns in *rect3* the intersection of *rect1* and *rect2*.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectIntersects

## SYNOPSIS

```
#include <graphics.h>
int RectIntersects(rect1,rect2)
     struct RECT *rect1,*rect2;
```

## Arguments

| | |
|---|---|
| <rect1>, <rect2> | The two *RECT* structures to be compared. |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns 1 if *rect1* and *rect2* intersect, otherwise 0.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectMerge

## SYNOPSIS

```
#include <graphics.h>
RectMerge(rect1,rect2,rect3)
     struct RECT *rect1,*rect2,*rect3;
```

### Arguments

| | |
|---|---|
| <rect1>, <rect2> | The two *RECT* structures to be compared. |
| <rect3> | A structure to hold the result. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Returns in *rect3* the minimum rectangle which contains both *rect1* and *rect2*.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# RectPrint

## SYNOPSIS

```
#include <graphics.h>
RectPrint(rect)
     struct RECT *rect;
```

## Arguments

| | |
|---|---|
| <rect> | A pointer to a *RECT* structure. |

## Returns

Nothing is returned.

## DESCRIPTION

Prints the address (in hexadecimal) and the values of the fields (in decimal) of the structure pointed to by the *rect* argument.

## ERRORS REPORTED

No errors are reported.

# ReleaseCursor

## SYNOPSIS

```
#include <graphics.h>
ReleaseCursor()
```

### Arguments

There are no arguments.

### Returns

Nothing is returned.

## DESCRIPTION

Tells the operating system to restore the cursor if it was removed due to a **ProtectCursor** call. This call should be used to match every **ProtectCursor** call.

## ERRORS REPORTED

No errors are reported.

# RestoreDisplayState

## SYNOPSIS

```
#include <graphics.h>
RestoreDisplayState(state)
     struct DISPSTATE *state;
```

## Arguments

| | |
|---|---|
| <state> | A pointer to a DISPSTATE structure which defines the display state to be restored. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

The state of the display is set to match the state which was previously stored in the state structure. This structure is not designed to be set up by hand, but rather to be filled in by a **SaveDisplayState** call.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# SaveDisplayState

## SYNOPSIS

```
#include <graphics.h>
SaveDisplayState(state)
     struct DISPSTATE *state;
```

### Arguments

| | |
|---|---|
| <state> | Pointer to a DISPSTATE structure in which the current state of the display will be stored. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

See the discussion of *DISPSTATE* in the conceptual introduction to the graphics library earlier in this manual.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# ScreenSaverEnable

## SYNOPSIS

```
#include <graphics.h>
ScreenSaverEnable(mode)
     int mode;
```

## Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

The screen saver is the function that blanks the screen after 10 minutes of keyboard inactivity. If the mode argument is *TRUE*, then the screen saver function is enabled. If *FALSE*, the screen saver is disabled. In either case, the previous setting is returned.

## ERRORS REPORTED

Display primitive failure.

# SetCPosition

## SYNOPSIS

```
#include <graphics.h>
SetCPosition(point)
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| <point> | A pointer to a POINT struct which contains the *x* and y screen coordinates to be used as the upper left corner of the cursor. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Displays the cursor at the *x,y* screen location defined by *point.*

## ERRORS REPORTED

Invalid (NULL) structure pointer

Graphics not initialized.

*x* or *y* outside range of screen.

Display primitive failure.

## NOTES

If the CursorTrack mode is enabled, this is the same as **SetMPosition**.

If *CursorTrack* mode is enabled, this is the same as **GetMPosition**.

Cursor location ("hot spot") is defined by the position (upper left corner) where the cursor is displayed, plus the offsets defined in the cursor form.

# SetCursor

## SYNOPSIS

```
#include <graphics.h>
SetCursor(curp)
     struct FORM *curp;
```

## Arguments

| | |
|---|---|
| <curp> | A pointer to a form structure which must define a 16 by 16 array of bits to be used as the cursor image. If this pointer is null, the default cursor image will be used. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Installs a new cursor. The bitmap and offsets of the 16 by 16 bit form defined by *curp* are copied into the system cursor form.

## ERRORS REPORTED

Invalid (*NULL*) structure pointer.

Invalid parameter in structure.

Display primitive failure.

# SetKBCode

## SYNOPSIS

```
#include <graphics.h>
SetKBCode(val)
     int val;
```

## Arguments

| | |
|---|---|
| <val> | An integer specifying the desired keyboard code. A zero specifies the event mechanism, while a one specifies ANSI mode. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

If set to ANSI mode, all keyboard activity is reported via ANSI strings of ASCII characters. If set to event mode, the ANSI terminal emulator is disabled and activity is reported only via the event mechanism. The event mechanism must be previously enabled or an error is returned.

## ERRORS REPORTED

Event mechanism not enabled.

Event primitive failure.

## NOTES

Turning on the event mechanism via the EventEnable call automatically sets the keyboard code to "event".

# SetMachineType

## SYNOPSIS

```
#include <graphics.h>
SetMachineType(value)
     int value;
```

### Arguments

| | |
|---|---|
| <value> | A hexadecimal value of the form 0x440x0000. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Sets the value (indicating a 4400 Series model number) to be returned by subsequent calls to GetMachineType.

## ERRORS REPORTED

Display primitive failure.

## SEE ALSO

GetMachineType, GetRealMachineType

# SetMBounds

## SYNOPSIS

```
#include <graphics.h>
SetMBounds(ulpoint,lrpoint)
     struct POINT *ulpoint;
     struct POINT *lrpoint;
```

## Arguments

| | |
|---|---|
| <ulpoint> | A pointer to a POINT structure which contains the *x* and *y* coordinates of the point to be used as the upper left corner of the mouse motion bounding box. |
| <lrpoint> | A pointer to a POINT structure which contains the *x* and *y* coordinates of the point to be used as the lower right corner of the mouse motion bounding box. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Sets the boundary of the region that contains the cursor.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Display primitive failure.

## NOTES

Both *x* and *y* must be in the range –32768 to 32767, inclusive.

If *CursorTrack* mode is enabled, this is the same as **GetMPosition**.

Cursor location ("hot spot") is defined by the position (upper left corner) where the cursor is displayed, plus the offsets defined in the cursor form.

# SetMPosition

## SYNOPSIS

```
#include <graphics.h>
SetMPosition(point)
     struct POINT *point;
```

### Arguments

| | |
|---|---|
| point | A pointer to a POINT struct which contains the *x* and *y* screen coordinates to be used as the mouse position. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Set mouse location to *x,y*.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Graphics haven't been initialized.

*X* or *y* outside range of screen.

Display primitive failure.

## NOTES

If CursorTrack mode is enabled, this is the same as SetCPosition.

If *CursorTrack* mode is enabled, this is the same as **GetMPosition**.

Cursor location ("hot spot") is defined by the position (upper left corner) where the cursor is displayed, plus the offsets defined in the cursor form.

# SetViewport

## SYNOPSIS

```
#include <graphics.h>
SetViewport(point)
     struct POINT *point;
```

## Arguments

| | |
|---|---|
| <point> | This is a pointer to a pair of short ints which define the *x* and *y* screen coordinates to be used as the upper left corner of the visible viewport. |

## Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Sets hardware to display with *x*, *y* of point as upper left corner.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Graphics haven´t been initialized.

*X* or *y* outside range of screen.

Display primitive failure.

# StringDraw

## SYNOPSIS

```
#include <graphics.h>
#include <font.h>
StringDraw(string,loc)
     char *string;
     struct POINT *loc;
```

### Arguments

| | |
|---|---|
| <string> | A *NULL* terminated array of characters. |
| <loc> | The location to start drawing. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *string* using the default font onto the screen starting at the point *loc*. The text bounding box (and clipping rectangle) is defined by the current viewport. The *bbS* combination rule is used. The value of *loc* is updated to reflect the end of the displayed text.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# StringDrawRawX

## SYNOPSIS

```
#include <graphics.h>
#include <font.h>
StringDrawRawX(string,loc,bbcom,font)
        char *string;
        struct POINT *loc;
        struct BBCOM *bbcom;
        struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <string> | A *NULL* terminated array of characters. |
| <loc> | The location to start drawing. |
| <bbcom> | Defines destination form, clipping rectangle, halftone form, and combination rule. |
| <font> | Font to use for displaying characters. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *string*, using *font* onto a form starting at the point *loc*. The *bbcom* argument should include the destination form, clipping rectangle, halftone form (optional) and combination rule.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Invalid parameter in structure.

## SEE ALSO

**StringDrawX**

# StringDrawX

## SYNOPSIS

```
#include <graphics.h>
#include <font.h>
StringDrawX(string,loc,bbcom,font)
      char *string;
      struct POINT *loc;
      struct BBCOM *bbcom;
      struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <string> | A *NULL* terminated array of characters. |
| <loc> | The location to start drawing. |
| <bbcom> | Defines destination form, destination rectangle, clipping rectangle, halftone form, and combination rule. |
| <font> | Font to use for displaying characters. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Draws *string*, using *font* onto a form starting at the point *loc*. The *bbcom* argument should include the destination form, destination rectangle (text bounding box), clipping rectangle, halftone form (optional) and combination rule. Line breaks, tabs, and scrolling within the text bounding box are managed automatically. The value of *loc* is updated to reflect the end of the displayed text.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

Invalid parameter in structure.

## SEE ALSO

**StringDrawRawX**

# StringWidth

## SYNOPSIS

```
#include <graphics.h>
include <font.h>
StringWidth(string,font);
        char *string;
        struct FontHeader *font;
```

### Arguments

| | |
|---|---|
| <string> | A *NULL* terminated array of characters. |
| <font> | Font to use for width calculation. |

### Returns

Returns zero if successful, otherwise –1 with *errno* set to the system or graphics error code.

## DESCRIPTION

Computes and returns the width in pixels required to draw *string* with *font*. This width is defined as the sum of the widths of the printable characters in *string*.

## ERRORS REPORTED

Invalid (NULL) structure pointer.

# TerminalEnable

## SYNOPSIS

```
#include <graphics.h>
int TerminalEnable(mode)
     int mode;
```

## Arguments

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

## Returns

Returns zero or positive if successful, otherwise –1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

If the mode argument is *TRUE*, then the terminal emulator is enabled and allowed to write characters on the screen. If *FALSE*, the terminal emulator is disabled. In either case, the previous setting is returned.

## ERRORS REPORTED

Display primitive failure.

# Appendix U
# 4400 Series ´C´ Reference Update

The following information was not available at printing of your *4400 Series ´C´ Reference Manual.* Please add the following discussions.

## cc

The ´C´ compiler, ***cc***, takes the additional option ***+Q***.

### Additional Options

| | |
|---|---|
| +Q | —Suppress quad word alignment on 68020 code generation* |

*68020 only

### Explanation Of Options

The 68020 ´C´ compiler, by default, aligns data structures on ***quad word*** (words consisting of four eight-bit bytes) boundaries. This, while allowing the 68020 to load and execute faster, causes "holes" in the data structures. The ***+Q*** option lets you suppress this alignment to allow close packing of data structures or compatilbility with data structures generated with non quad-aligned compilers, such as 68000 or 68010 compilers.

## Floating Point Processor Signals

The floating point processor can generate signals or interrupts when it finds exceptional conditions. By default, the operating system ignores these signals. If you want or need to use the floating point processor signals, it is your responsibility to code the signal handling routines.

The operating system examines a *floating-point signal bit* in the binary header of executable files to enable or disable floating point signal processing. This bit is set (or by default left unset) by the loader. To compile code that will do floating point signal processing, you must either enable floating point processing after compilation by using *headset*, or pass the loader option ***+q*** to *load* during compilation and loading.

To enable floating point processing after compilation, type:

```
        ++ headset +I
        ++
```

To compile a program, *test.c*, with floating point signals enabled, type:

```
        ++ cc test.c +x=q
        ++
```

*CAUTION*

*When you use mixed-mode arithmetic (floating point and integer) you may loose precision in your results. If you are using a 68020 based machine, the details of the floating point coprocessor operation can be found in the **MC68881 Floating-Point Coprocessor User's Manual** published by the Motorola Corp.*

# VideoNormal

## SYNOPSIS

```
#include <graphics.h>
int VideoNormal(mode)
    int mode;
```

## ARGUMENTS

| | |
|---|---|
| <mode> | An integer which should be either *TRUE* (non-zero) or *FALSE* (zero). |

## RETURNS

Returns zero or positive if successful, otherwise −1 with *errno* set to the system or graphics error code. If the returned value is zero, then the mode was previously false or disabled, if positive the mode was true or enabled.

## DESCRIPTION

If the mode argument is *TRUE*, then the display is set to normal video (black on white). If the argument is *FALSE*, the display is set to inverse video (white on black). In either case, the previous setting is returned.

## ERRORS REPORTED

Display primitive failure.