Microsoft®

MICROSOFT® C

FOR THE MS-DOS® OPERATING SYSTEM

RUN-TIME LIBRARY REFERENCE

Document No. 410840017-500-R04-0887

TABLE OF CONTENTS

Part 1 ◇ Overview

Part 2 ◇ Reference

Appendixes

Figures

Tables

PART 1

OVERVIEW

CHAPTERS

PART 1

OVERVIEW

The first part of this manual provides information common to all of the run-time library functions.

Here you'll find descriptions of the common attributes of the run-time library, definitions for global variables, data types, and include files, and useful background information on the different categories of run-time routines.

CHAPTER 1

INTRODUCTION

1.1 About the C Library

The Microsoft® C Run-Time Library is a set of more than 200 predefined functions and macros designed for use in C programs. The run-time library makes programming easier by providing the following:

1. An interface to operating-system functions (such as opening and closing files)
2. Fast and efficient functions to perform common programming tasks (such as string manipulation), sparing the programmer the time and effort needed to write such functions

The run-time library is especially important in C programming because C programmers rely on the library for basic functions not provided by the language. These functions include, among others, input and output, storage allocation, and process control.

The functions in the Microsoft C Run-Time Library have been designed to maintain maximum compatibility between MS-DOS® and XENIX® or UNIX™ systems. Throughout this manual, references to XENIX systems encompass UNIX and other UNIX-like systems as well.

Most of the functions in the C run-time library for MS-DOS operate compatibly with functions having the same names in the C run-time library for XENIX operating systems. If you are interested in portability, see Appendix B, "Common Libraries." This appendix lists the functions of the run-time library that are specific to MS-DOS and describes differences (if any) between the operation of functions with the same names on XENIX and MS-DOS.

For additional compatibility, the math functions of the Microsoft C Run-Time Library have been extended to provide exception handling in the same manner as UNIX System V math functions.

The library is also designed for compatibility with the Draft Proposed American National Standard—Programming Language C (ANSI C), except for the internationalization functions. The functions that conform to the ANSI C standard are also listed in Appendix B.

For programmers who are interested in taking advantage of the specific features of MS-DOS, the library includes MS-DOS interface functions.

These functions allow MS-DOS system calls and interrupts to be invoked from a C program. The library also contains console input and output functions to allow efficient reading and writing from the user's console.

To take advantage of the compiler's type-checking capabilities, the include files that accompany the run-time library have been expanded. In addition to the definitions and declarations required by library functions and macros, the include files now contain function declarations with argument-type lists. The argument-type lists enable type checking for calls to library functions. This feature can be extremely helpful in detecting subtle program errors resulting from type mismatches between actual and formal arguments to a function.

To provide argument-type lists for all run-time functions, several new include files have been added to the list of standard include files for the C run-time library. The names of the new include files have been chosen to maintain as much compatibility as possible with the proposed ANSI Standard for C and with XENIX and UNIX names.

1.2 About This Manual

The *Microsoft C Run-Time Library Reference* describes the contents of the Microsoft C Run-Time Library. The manual assumes that you are familiar with the C language and with MS-DOS. It also assumes that you know how to compile and link C programs on your MS-DOS system and that you can set up a compiler and linker environment using environment variables. If you have questions about compiling, linking, or setting up an environment, see your compiler guide. If you have questions about the C language, see the *Microsoft C Language Reference.*

Note

> Since MS-DOS and PC-DOS are essentially the same operating system, this manual uses the term MS-DOS to refer to both systems, except in those cases where the distinction is significant.

This manual has two major parts. Part 1, "Overview," gives an introduction to the C run-time library. It discusses general rules that apply to the run-time library as a whole and summarizes the elements of the run-time library.

Part 2, "Reference," gives descriptions of the run-time routines in alphabetical order for quick reference. Once you have familiarized yourself with the library rules and procedures, you will likely use Part 2 of this manual most often.

The remaining chapters of Part 1 are as follows:

Chapter 2, "Using C Library Routines," gives general rules for understanding and using C library routines and mentions special considerations that apply to certain routines. It is recommended that you read this chapter before using the run-time library; you may also want to turn to Chapter 2 when you have questions about library procedures.

Chapter 3, "Global Variables and Standard Types," describes variables and types that are declared and defined in the run-time library and used by the library routines. This chapter also provides a cross-reference to the include file that defines or declares these variables and types. You may find them useful in your own routines. They are also described in the reference pages for the routines that use them.

Chapter 4, "Run-Time Routines by Category," breaks down the run-time library routines by category, lists the routines that fall into each category, and discusses considerations that apply to each category as a whole. The chapter complements the reference section, making it easy to locate routines by task. Once you decide on the routines you want, simply turn to the appropriate reference pages in Part 2 for a detailed description.

Chapter 5, "Include Files," summarizes the contents of each include file provided with the run-time library.

The appendixes at the back of the binder provide more detailed information about error messages and about XENIX-compatible routines. Appendix A, "Error Messages," describes the error values and messages that can appear when using library routines. Appendix B, "Common Libraries," lists routines of the MS-DOS C library that operate compatibly both with routines of the same name on XENIX and UNIX systems and with routines that conform to the Draft Proposed American National Standard—Programming Language C. Appendix B also describes differences (if any) between the DOS and XENIX versions of the routines and discusses common global variables and include files.

The remainder of this chapter describes the notational conventions used throughout the manual.

1.3 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Meaning
keywords, **routines,** **include files**	C keywords, such as **double** and **char**, are set in bold type to distinguish them from ordinary identifiers and text. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown. The names of run-time library routines, include files, global variables, standard types, constants, and identifiers used by the C library are also set in this font to emphasize that these names are reserved by the run-time library. For example, the routine name **strcpy** appears in this font; so does the include file **stdio.h**.
ENVIRONMENT VARIABLES, MS-DOS COMMANDS	Bold capital letters are used for the names of environment variables (such as **TZ** and **PATH**) and MS-DOS commands (such as **SET** and **PATH**). However, on MS-DOS you are not required to use capital letters for these variables and commands.
placeholders	Italics are used for the names of arguments to library routines. In an actual program, a specific name or value replaces the italicized argument name. For example, in **double atof(***string***);** the argument *string* is italicized to indicate that this is the general form for the **atof** routine. In an actual program, the user supplies a particular argument for the placeholder *string.* Occasionally, italics are used to emphasize particular words in the text.
`Examples`	Programming examples are displayed in a special typeface to resemble the output on your screen or the output of commonly used computer printers. Program fragments and variables quoted within regular text also appear in this format, as do error messages.

`User input`

Some examples show both program output and user input; in these cases, input is shown in a darker font. In the following example, **`.5`** is entered by the user in response to the prompt `Cosine value =:`

```
Cosine value = .5
Arc cosine of 0.500000 = 1.047198
```

`Missing`
.
.
.
`code`

Vertical ellipsis dots are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipsis dots between the two statements indicate that intervening program lines occur but are not shown:

```
int x, y;
.
.
.
y = abs(x);
```

Repeating
elements ...

Horizontal ellipsis dots following an item indicate that more items having the same form may appear. For instance,

= { *expression* ⟦,*expression*⟧...}

indicates that more expressions, separated by commas, may appear between the braces ({ }).

⟦optional items⟧

Double brackets enclose optional arguments in the specification for each library routine. For example, in

int open(*path, oflag*⟦, *pmode*⟧**);**

the double brackets around *pmode* indicate that this argument is optional and that, when given, *pmode* must be separated from the previous argument by a comma.

Arrays [ ]

Subscripts [ ]

Single brackets appear in syntax descriptions and examples containing arrays and subscript expressions. The C language also uses brackets for array declarations and subscript expressions. To illustrate,

```
char *args[4];
```

is an example showing the declaration of a four-element array; the brackets around 4 are a required part of the C language.

“Defined terms”	Quotation marks set off terms defined in the text. For example, the term “token” appears in quotation marks when it is defined. Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `"  "` rather than “ ”. For example, `"abc"` is a C string.
KEY+KEY	Small capital letters are used for the names of keys and key sequences, such as CTRL+C.

CHAPTER 2

USING C LIBRARY ROUTINES

2.1 Introduction

To use a C library routine, simply call it in your program, just as if the routine were defined in your program. The C library functions are stored in compiled form in the library files that accompany your C compiler software.

At link time, your program must be linked with the appropriate C library file or files to resolve the references to the library functions and provide the code for the called library functions. The procedures for linking with the C library are discussed in detail in Chapter 4 of the *Microsoft C Optimizing Compiler User's Guide.*

In most cases you must prepare for the call to the run-time library function by performing one or both of the following steps:

1. Include a given file in your program. Many routines require definitions and declarations that are provided by an include file.
2. Provide declarations for library functions that return values of any type but **int**. The compiler expects all functions to have **int** return type unless declared otherwise. You can provide these declarations by including the C library file containing the declarations or by explicitly declaring the functions within your program.

These are the minimum steps required; you may also want to take other steps, such as enabling type checking for the arguments in function calls.

The remainder of this chapter discusses the preparation procedures for using run-time library routines and special rules (such as file-name and path-name conventions) that may apply to some routines.

2.2 Identifying Functions and Macros

The words "function" and "routine" are used interchangeably throughout this manual, and in fact most of the routines in the C run-time library are C functions; that is, they consist of compiled C statements. However, some routines are implemented as "macros." A macro is an identifier defined with the C preprocessor directive **#define** to represent a value or expression. Like a function, a macro can be defined to take zero or more arguments, which replace formal parameters in the macro definition. Defining and using macros are discussed in detail in Chapter 8 of the *Microsoft C Language Reference.*

The macros defined in the C run-time library behave like functions: they take arguments and return values, and they are invoked in a similar manner. The major advantage of using macros is faster execution time; their definitions are expanded (replaced by their definitions) in the preprocessing stage, eliminating the overhead required for a function call. However, because macros are expanded before compilation, they can increase the size of a program, particularly when the macro appears several times in the program. Unlike a function, which is defined only once regardless of how many times it is called, each occurrence of a macro is expanded. Functions and macros thus offer a trade-off between speed and size. In several cases, the C library allows you to choose by providing both macro and function versions of the same library routine.

Some important differences between functions and macros are described in the following list:

1. Some macros may treat arguments with side effects incorrectly when the macro is defined so that arguments are evaluated more than once. See the example that follows this list.
2. A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address, as a function identifier does. You cannot, therefore, use a macro identifier in contexts requiring a pointer. For instance, if you give a macro identifier as an argument in a function call, the *value* represented by the macro is passed; if you give a function identifier as an argument in a function call, the *address* of the function is passed.
3. Since macros are not functions, they cannot be declared, nor can pointers to macro identifiers be declared. Thus, type checking cannot be performed on macro arguments. The compiler does, however, detect cases where the wrong number of arguments is specified for the macro.
4. The library routines implemented as macros are defined through preprocessor directives in the library include files. To use a library macro, you must include the appropriate file, or the macro will be undefined.

The routines that are implemented as macros are noted in the reference section of this manual. You can examine particular macro definitions in the corresponding include file to determine whether arguments with side effects will cause problems.

■ Example

```
#include <ctype.h>

int a = 'm';
a = toupper(a++);
```

This example uses the **toupper** routine from the standard C library. The **toupper** routine is implemented as a macro; its definition in **ctype.h** is as follows:

```
#define toupper(c)   ( (islower(c)) ? _toupper(c) : (c) )
```

The definition uses the conditional operator (**? :**). In the conditional expression, the argument `c` is evaluated twice: once to determine whether or not it is lowercase, and once to return the appropriate result. This causes the argument `a++` to be evaluated twice, thus increasing `a` twice rather than once. As a result, the value operated on by **islower** differs from the value operated on by **_toupper**.

Not all macros have this effect; you can determine whether a macro will handle side effects properly by examining the macro definition before using it.

2.3 Including Files

Many run-time routines use macros, constants, and types that are defined in separate include files. To use these routines, you must incorporate the specified file (using the preprocessor directive **#include**) into the source file being compiled.

The contents of each include file are different, depending on the needs of specific run-time routines. However, in general, include files contain combinations of the following:

- Definitions of manifest constants

  For example, the constant **BUFSIZ**, which determines the hardware-dependent size of buffers for buffered input and output operations, is defined in **stdio.h**.

- Definitions of types

  Some run-time routines take data structures as arguments or return values with structure types. Include files set up the required structure definitions. For example, most stream input and output operations use pointers to a structure of type **FILE**, defined in **stdio.h**.

- Two sets of function declarations

  The first set of declarations gives return types and argument-type lists for run-time functions; the second set declares only the return type. Declaring the return type is required for any function that returns a value with type other than **int**. (See Section 2.4, "Declaring Functions.") The presence of an argument-type list enables type checking for the arguments in a function call. See Section 2.6, "Argument-Type Checking," for a discussion of this option.

- Macro definitions

  Some routines in the run-time library are implemented as macros. The definitions for these macros are contained in the include files. To use one of these macros, you must include the appropriate file.

The include file or files needed by each routine can be found in the reference section of this manual.

2.4 Declaring Functions

Whenever you call a library function that returns any type of value but an **int**, you should make sure that the function is declared before it is called. The easiest way to do this is to include the file containing declarations for that function, causing the appropriate declarations to be placed in your program.

Two sets of function declarations are provided in each include file. The first set declares both the return type and the argument-type list for the function. This set is included only when you enable argument-type checking, as described in Section 2.6. Use of the type-checking feature is highly recommended, since type mismatches between a function's arguments and formal parameters can cause serious and possibly hard-to-detect errors.

The second set of function declarations declares only the return type. This set is included when argument type checking is *not* enabled.

Your program can contain more than one declaration of the same function as long as the declarations are compatible. This is an important feature to remember if you have older programs whose function declarations do not contain argument-type lists. For instance, if your program contains the declaration

```
char *calloc( );
```

you can also include the following declaration:

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

Although the two declarations are not identical, they are compatible, so no conflict occurs.

If you wish, you can provide your own function declarations instead of using the declarations in the library include files. However, you should consult the declarations in the include files to make sure that your declarations are correct.

2.5 Stack Checking on Entry

Stack checking means that, on entry to a routine, the stack is first checked to determine whether or not there is room for the local variables used by that routine. If there is, space is allocated by adjusting the stack pointer. Otherwise, a "stack overflow" run-time error occurs. If stack checking is disabled, the compiler assumes there is enough stack space. If in fact there is not sufficient space on the stack, you may overwrite memory locations in the data segment and receive no warning.

Typically, only functions with large local variable requirements (more than about 150 bytes) have stack checking enabled, since there is enough free space between the stack and data segments to handle functions with smaller requirements. If the function is called many times, stack checking will slow down execution slightly.

The following library functions have stack checking enabled:

execvp
execvpe
printf
fprintf
sprintf
vprintf
scanf
fscanf
sscanf
spawnvp
spawnvpe
system

2.6 Argument-Type Checking

Microsoft C offers a type-checking feature for the arguments in a function call. Type checking is performed whenever an argument-type list is present in a function declaration and the declaration appears before the definition or use of the function in a program. The form of the argument-type list and the type-checking method are discussed in full in Chapter 7 of the *Microsoft C Language Reference.*

For functions that you write yourself, you are responsible for setting up argument-type lists to invoke type checking. You can also use the **/Zg** command-line option to cause the compiler to generate a list of function declarations for all functions defined in a particular source file; the list can then be incorporated into your program. See your compiler guide for details on using the **/Zg** option.

For functions in the C run-time library, type checking is always enabled. Only limited type checking can be performed on functions that take a variable number of arguments. The following run-time functions are affected by this limitation:

- In calls to **cprintf**, **cscanf**, **printf**, and **scanf**, type checking is performed only on the first argument: the format string.
- In calls to **fprintf**, **fscanf**, **sprintf**, and **sscanf**, type checking is performed on the first two arguments: the file or buffer and the format string.
- In calls to **open**, only the first two arguments are type checked: the path name and the open flag.
- In calls to **sopen**, the first three arguments are type checked: the path name, the open flag, and sharing mode.
- In calls to **execl**, **execle**, **execlp**, and **execlpe**, type checking is performed on the first two arguments: the path name and the first argument pointer.
- In calls to **spawnl**, **spawnle**, **spawnlp**, and **spawnlpe**, type checking is performed on the first three arguments: the mode flag, the path name, and the first argument pointer.

2.7 Error Handling

When calling a function, it is a good idea to provide for detection and handling of error returns, if any. Otherwise, your program may produce unexpected results.

For run-time library functions, you can determine the expected return value from the return-value discussion on each library page. In some cases no established error return exists for a function. This usually occurs when the range of legal return values makes it impossible to return a unique error value.

The description of some functions in Part 2 indicates that when an error occurs, a global variable named **errno** is set to a value indicating the type of error. Note that you cannot depend on **errno** being set unless the description of the function explicitly mentions the **errno** variable.

When using functions that set **errno**, you can test the **errno** values against the error values defined in **errno.h**, or you can use the **perror** or **strerror** functions. If you want to print the system error message to standard error (**stderr**), use **perror**; if you want to store the error message in a string for later use in your program, use **strerror**. For a list of **errno** values and the associated error messages, see Appendix A, "Error Messages."

When you use **errno**, **perror**, and **strerror**, remember that the value of **errno** reflects the error value for the last call that set **errno**. To prevent misleading results, you should always test the return value before accessing **errno**, to verify that an error actually occurred. Once you determine that an error has occurred, use **errno** or **perror** immediately. Otherwise, the value of **errno** may be changed by intervening calls.

The math functions set **errno** upon error in the manner described on the reference page for each math function in Part 2. Math functions handle errors by invoking a function named **matherr**. You can choose to handle math errors differently by writing your own error function and naming it **matherr**. When you provide your own **matherr** function, that function is used in place of the run-time library version. You must follow certain rules when writing your own **matherr** function, as outlined in the reference section.

You can check for errors in stream operations by calling the **ferror** function. The **ferror** function detects whether the error indicator has been set for a given stream. When the stream is closed or rewound, the error indicator is cleared automatically; or you can reset it by calling the **clearerr** function.

Errors in low-level input and output operations cause **errno** to be set.

The **feof** function tests for end-of-file on a given stream. An end-of-file condition in low-level input and output can be detected with the **eof** function or when a **read** operation returns 0 as the number of bytes read.

2.8 File Names and Path Names

Many functions in the run-time library accept strings representing path names and file names as arguments. The functions process the arguments and pass them to the operating system, which is ultimately responsible for creating and maintaining files and directories. Thus, it is important to keep in mind not only the C conventions for strings, but also the operating system rules for file names and path names and the differences between MS-DOS and XENIX rules. There are three considerations:

1. Case sensitivity
2. Subdirectory conventions
3. Delimiters for path-name components

The C language is case sensitive, meaning that it distinguishes between uppercase and lowercase letters. The MS-DOS operating system is not case sensitive. When accessing files and directories on MS-DOS, you cannot use case differences to distinguish between identical names. For example, the names "FILEA" and "fileA" are equivalent and refer to the same file.

Portability considerations may also affect how you choose file names and path names. For instance, if you plan to port your code to a XENIX system, you should take the XENIX naming conventions into account. Unlike MS-DOS, XENIX is case sensitive. Thus, the following two directives are equivalent on MS-DOS but not on XENIX:

```
#include <STDIO.H>
#include <stdio.h>
```

To produce portable code, you should use the name that works correctly on XENIX, since either case works on MS-DOS.

The convention of storing some include files in a subdirectory named **sys** is also a XENIX convention. The convention is adopted in this manual, which includes the **sys** subdirectory in the specification for the appropriate include files. If you're not concerned with portability, you can disregard this convention and set up your include files accordingly. If you are concerned with portability, using the **sys** subdirectory can make portability between XENIX and MS-DOS easier.

The MS-DOS and XENIX operating systems differ in the handling of path-name delimiters. XENIX uses the forward slash (/) to delimit the components of path names, while MS-DOS ordinarily uses the backslash (\). However, MS-DOS recognizes the forward slash as a delimiter in situations where a path name is expected. Thus, you may use either a backslash or a forward slash in MS-DOS path names within C programs, as long as the context is unambiguous and a path name is clearly expected.

Note

> In C strings, the backslash is an escape character. It signals that a special escape sequence follows. If an ordinary character follows the backslash, the backslash is disregarded and the character is printed. Thus, the sequence "\\" is required to produce a single backslash in a C string. (See Chapter 2 of the *Microsoft C Language Reference* for a full discussion of escape sequences.)

For most of the functions in the run-time library, you may use either a forward slash or a backward slash as a delimiter whenever a path-name argument is required. If you are concerned with portability to XENIX, you should use the forward slash.

However, the exceptions to the rule are important. The following functions accept string arguments that are not known in advance to be path names (they may be path names, but are not required to be). In these cases, the arguments are treated as C strings, and the following special rules apply:

- In the **exec** and **spawn** families of functions, you pass the name of a program to be executed as a child process and then pass strings representing arguments to the child process. The path name of the program to be executed as the child process can use either forward slashes or backslashes as delimiters, since a path-name argument is expected. However, it is recommended that you use backslashes in any path-name arguments to the child process, since the program being executed as the child process may simply expect a string argument that is not necessarily a path name.

- In the **system** call, you pass a command to be executed by MS-DOS; this command may or may not include a path name.

In these cases, only the backslash (\) separator should be used as a path-name delimiter. The forward slash (/) will not be recognized.

When you want to pass a path-name argument to the child process in an **exec** or **spawn** call, or when you use a path name in a **system** call, you must use the double-backslash sequence (\\) to represent a single path-name delimiter.

■ Examples

```
result = system("DIR B:\\TOP\\DOWN");
```

In the example above, double backslashes must be used in the call to **system** to represent the path name `B:\TOP\DOWN`. Note that not all calls to **system** use a path name; for example,

```
result = system("DIR");
```

does not contain a path name.

```
spawnl(P_WAIT, "bin/show", "show", "sub", "bin\\tell", NULL);
```

In the above example, the **spawnl** function is used to execute the file named `show.exe` in the `bin` subdirectory. Since a path name is expected as the second argument, the forward slash can be used. (A double backslash would also be acceptable.) The first two arguments passed to `show.exe` are the strings `show` and `sub`. The third argument is a string representing a path name. Since this argument does not require a path name, the sequence \\ must be used to represent a single backslash between `bin` and `tell`.

2.9 Binary and Text Modes

Most C programs use one or more data files for input and output. Under MS-DOS, data files are ordinarily processed in "text mode." In text mode, carriage-return–line-feed combinations are translated into a single line-feed character on input. Line-feed characters are translated to carriage-return–line-feed combinations on output.

In some cases you may want to process files without making these translations. In binary mode, carriage-return–line-feed translations are suppressed.

You can control the translation mode for program files in the following ways:

- To process a few selected files in binary mode, while retaining the default text mode for most files, you can specify binary mode when you open the selected files. The **fopen** function opens a file in binary mode when the letter **b** is specified in the access *type* string for the file. If you use the **open** function, you can specify the **O_BINARY** flag in the *oflag* argument to cause the file to be opened in binary mode. For more information, see the reference pages for these functions in the reference section of this manual.

- To process most or all files in binary mode, you can change the default mode to binary. The global variable **_fmode** controls the default translation mode. When **_fmode** is set to **O_BINARY**, the default mode is binary, except for **stdaux** and **stdprn**, which are opened in binary mode by default. The initial setting of **_fmode** is text, by default.

  You can change the value of **_fmode** in one of two ways. First, you can link with the file **BINMODE.OBJ** (supplied with your compiler software). Linking with **BINMODE.OBJ** changes the initial setting of **_fmode** to **O_BINARY**, causing all files except **stdin**, **stdout**, and **stderr** to be opened in binary mode. This option is described in Chapter 3 of the *Microsoft C Optimizing Compiler User's Guide.*

  Second, you can change the value of **_fmode** directly by setting it to **O_BINARY** in your program. This has the same effect as linking with **BINMODE.OBJ**.

  You can still override the default mode (now binary) for particular files by opening them in text mode. The **fopen** function opens a file in text mode when the letter **t** is specified in the access *type* string for the file. If you use the **open** function, you can specify the **O_TEXT** flag in the *oflag* argument to cause the file to be opened in text mode. For more information, see the reference pages for these functions in Part 2.

- The **stdin**, **stdout**, and **stderr** streams are opened in text mode by default; **stdaux** and **stdprn** are opened in binary mode. To process **stdin**, **stdout**, or **stderr** in binary mode instead, or to process **stdaux** or **stdprn** in text mode, use the **setmode** function. This function can also be used to change the mode of a file after it has been opened. The **setmode** function takes two arguments, a file handle and a translation-mode argument, and sets the mode of the file accordingly.

2.10 MS-DOS Considerations

The use of some functions in the run-time library is affected by the version of MS-DOS you are using. These functions are listed and described below:

Function	**Restrictions**
dosexterr **locking** **sopen**	These three functions are effective only on MS-DOS Versions 3.0 and later. The **sopen** function opens a file with file-sharing attributes; this function should be used instead of **open** when you want a file to have such attributes. The **locking** function locks all or part of a file from access by other users. The **dosexterr** function provides error handling for system call 59H (Get Extended Error) in MS-DOS Versions 3.0 and later.
dup **dup2**	In certain cases, using the **dup** and **dup2** functions on versions of MS-DOS earlier than 3.0 may cause unexpected results. When you use **dup** or **dup2** to create a duplicate file handle for **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** under versions of MS-DOS earlier than 3.0, calling the **close** function with either handle causes errors in later I/O operations that use the other handle. Under MS-DOS Versions 3.0 and later, the **close** function is handled correctly and does not cause later errors.
exec **spawn**	When using the **exec** and **spawn** families of functions under versions of MS-DOS earlier than 3.0, the value of the *arg0* argument (or *argv*[**0**] to the child process) is not available to the user; the character "C" is stored in that position instead. Under MS-DOS Versions 3.0 and later, the complete command path is stored in *arg0*.

To write programs that will run on all versions of MS-DOS, you can use the **_osmajor** and **_osminor** variables (discussed in Chapter 3, "Global Variables and Standard Types"). These variables allow you to ascertain the current operating-system version number and take the appropriate action based on the result of the test.

■ Example

```
unsigned char _osmajor;
.
.
.
if (_osmajor < 3)
        open("TEST.DAT", O_RDWR);
else
        sopen("TEST.DAT", O_RDWR, SH_DENYWR);
```

In the above example, the global variable **_osmajor** is tested to determine whether the file TEST.DAT should be opened using the **open** function (under versions of MS-DOS earlier than 3.0) or the **sopen** function (MS-DOS Versions 3.0 and later).

2.11 Floating-Point Support

The math functions supplied in the C run-time library require floating-point support to perform calculations with real numbers. This support can be provided by the floating-point libraries that accompany your compiler software or by an 8087 or 80287 coprocessor. (For information on selecting and using a floating-point library with your program, see your compiler guide. The names of the functions that require floating-point support are listed below:

acos
asin
atan
atan2
atof
bessel[2]
cabs
ceil
_clear87[1]
_control87[1]
cos
cosh
dieeetomsbin
difftime
dmsbintoieee
ecvt
exp
fabs
fcvt
fieeetomsbin
floor
fmod
fmsbintoieee
_fpreset
frexp
gcvt
hypot
ldexp
log
log10
modf
pow
sin
sinh
sqrt
_status87[1]
strtod
tan
tanh

In addition, the **printf** family of functions (**cprintf**, **fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, and **vsprintf**) requires support for floating-point input and output if used to print floating-point values.

The C compiler tries to detect whether floating-point values are used in a program so that supporting functions are loaded only if required. This behavior saves a considerable amount of space for programs that do not require floating-point support.

---

[1] Not available with the **/FPa** compiler option

[2] The **bessel** function does not correspond to a single function, but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

When you use a floating-point type character in the format string for a **printf** or **scanf** call (**cprintf**, **fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, **vsprintf**, **cscanf**, **fscanf**, **scanf**, or **sscanf**), make sure that you specify floating-point values or pointers to floating-point values in the argument list to correspond to any floating-point type characters in the format string. The presence of floating-point arguments allows the compiler to detect floating-point values. If a floating-point type character is used to print an integer argument, for example, floating-point values will not be detected because the compiler does not actually read the format string used in the **printf** and **scanf** functions. For instance, the following program produces an error at run time:

```
main( ) /* THIS EXAMPLE PRODUCES AN ERROR */
 {
  long f = 10L;
  printf("%f", f);
 }
```

In the preceding example, the functions for floating-point I/O are not loaded for the following reasons:

- No floating-point arguments are given in the call to **printf**.
- No floating-point values are used anywhere else in the program.

As a result, the following error occurs:

```
Floating point not loaded
```

The following is a corrected version of the above call to **printf**:

```
main( ) /* THIS EXAMPLE WORKS JUST FINE */
 {
  long f = 10L;
  printf("%f", (double) f);
 }
```

This version corrects the error by casting the long integer value to **double**.

2.12 Using Huge Arrays with Library Functions

In programs that use small, compact, medium, and large memory models, Microsoft C allows you to use arrays exceeding the 64K (kilobyte) limit of physical memory in these models by explicitly declaring the arrays as

huge. (See your compiler guide for a complete discussion of memory models and the **near**, **far**, and **huge** keywords.) However, you cannot generally pass **huge** data items as arguments to C library functions. In the case of small and medium models, where the default size of a data pointer is **near** (16 bits), the only routines that accept huge pointers are **halloc** and **hfree**. In the compact-model library used by compact-model programs, and in the large-model library used by both large-model and huge-model programs, only the functions listed below use argument arithmetic that works with **huge** items:

bsearch
fread
fwrite
halloc
hfree
lfind
lsearch
memccpy
memchr
memcmp
memcpy
memicmp
memset
qsort

With this set of functions, you can read from, write to, search, sort, copy, initialize, compare, or dynamically allocate and free **huge** arrays; a **huge** array can be passed without difficulty to any of these functions in a compact-, large-, or huge-model program.

There is a semantic difference between the function and intrinsic versions of the **memset**, **memcpy**, and **memcmp** library routines. The function versions of these routines support huge pointers in compact and large model, but the intrinsic versions do not support huge pointers.

CHAPTER 3

GLOBAL VARIABLES AND STANDARD TYPES

3.1 Introduction

The C run-time library contains definitions for a number of variables and types used by library routines. You can access these variables and types by including in your program the files in which they are declared or by giving appropriate declarations in your program, as shown in the following sections.

3.2 _amblksiz

unsigned _amblksiz;

The **_amblksiz** variable can be used to control the amount of memory space in the heap that is used by C for dynamic memory allocation. This variable is declared in the include file **malloc.h**.

The first time your program calls one of the dynamic-memory-allocation functions, such as **calloc** or **malloc**, it asks the operating system for an initial amount of heap space that is typically much larger than the amount of memory requested by **calloc** or **malloc**. This amount is indicated by **_amblksiz**, whose default value is 8K (8192 bytes). Subsequent memory allocations are allotted from this 8K of memory, resulting in fewer calls to the operating system when many relatively small items are being allocated. C calls the operating system again only if the amount of memory used by dynamic memory allocations exceeds the currently allocated space.

If the requested size in your C program is greater than **_amblksiz**, multiple blocks, each of size **_amblksiz**, are allocated until the request is satisfied; since the amount of heap space allocated is more than the amount requested, subsequent allocations can cause fragmentation of heap space. You can control this fragmentation by using **_amblksiz** to change the default "memory chunk" to whatever value you like, as in the following example:

```
_amblksiz = 2000;
```

Since the heap allocator always rounds the MS-DOS request to the nearest power of two greater than or equal to **_amblksiz**, the preceding statement causes the heap allocator to reserve memory in the heap in multiples of 2K (2048 bytes).

Note that adjusting the value of **_amblksiz** affects only far-heap allocation (for example, standard **malloc** calls in compact, large, and huge memory models and **_fmalloc** calls in small and medium memory models). Adjusting this value has no effect on **halloc** or **_nmalloc** in any memory model.

3.3 daylight, timezone, tzname

int daylight;
long timezone;
char *tzname[2];

The **daylight**, **timezone**, and **tzname** variables are used by several of the time and date functions to make local-time adjustments and are declared in the include file **time.h**. The values of the variables are determined by the setting of an environment variable named **TZ**.

You can control local-time adjustments by setting the **TZ** environment variable. The value of the environment variable **TZ** must be a three-letter time zone, followed by a possibly signed number giving the difference in hours between Greenwich mean time and local time. The number is positive moving west from Greenwich, negative moving east. The number may be followed by a three-letter daylight-saving-time (DST) zone. For example, the command

```
SET TZ=EST5EDT
```

specifies that the local time zone is EST (Eastern standard time), that local time is five hours earlier than Greenwich mean time, and that EDT is the name of the time zone when daylight saving time is in effect. Omitting the DST zone, as shown below, means that daylight time is never in effect:

```
SET TZ=EST5
```

When you call the **ftime** or **localtime** function, the values of the three variables **daylight**, **timezone**, and **tzname** are determined from the **TZ** setting. The **daylight** variable is given a nonzero value if a DST zone is present in the **TZ** setting; otherwise, **daylight** is 0. The **timezone** variable is assigned the difference in seconds (calculated by converting the hours given in the **TZ** setting) between Greenwich mean time and local time. The first element of the **tzname** variable is the string value of the three-letter time zone from the **TZ** setting; the second element is the string value of the DST zone. If the DST zone is omitted from the **TZ** setting, **tzname**[1] is an empty string.

If you do not explicitly assign a value to **TZ** before calling **ftime** or **localtime**, the following default setting is used:

```
PST8PDT
```

The **ftime** and **localtime** functions call another function, **tzset**, to assign values to the three global variables from the **TZ** setting. You can also call **tzset** directly if you like; see the **tzset** reference page in Part 2 of this manual for details.

3.4 _doserrno, errno, sys_errlist, sys_nerr

int _doserrno;
int errno;
char *sys_errlist[ ];
int sys_nerr;

The **errno**, **sys_errlist**, and **sys_nerr** variables are used by the **perror** function to print error information and are declared in the include file **stdlib.h**. When an error occurs in a system-level call, the **errno** variable is set to an integer value to reflect the type of error. The **perror** function uses the **errno** value to look up (index) the corresponding error message in the **sys_errlist** table. The value of the **sys_nerr** variable is defined as the number of elements in the **sys_errlist** array. For a listing of the **errno** values and the corresponding error messages, see Appendix A, "Error Messages."

The **errno** values on MS-DOS are a subset of the values for **errno** on XENIX systems. Therefore, the value assigned to **errno** in case of error does not necessarily correspond to the actual error code returned by an MS-DOS system call. Instead, the actual MS-DOS error codes are mapped onto the **perror** values. If you want to access the actual MS-DOS error code, use the **_doserrno** variable. When an error occurs in a system call, the **_doserrno** variable is assigned the actual error code returned by the corresponding MS-DOS system call.

In general, you should use **_doserrno** only for error detection in operations involving input and output, since the **errno** values for input and output errors have MS-DOS error-code equivalents. Not all of the error values available for **errno** have exact MS-DOS error-code equivalents, and some may have no equivalents, causing the value of **_doserrno** to be undefined.

3.5 _fmode

int _fmode;

The **_fmode** variable controls the default file-translation mode. It is declared in **stdlib.h**. By default, the value of **_fmode** is 0, causing files to be translated in text mode (unless specifically opened or set to binary mode). When **_fmode** is set to **O_BINARY**, the default mode is binary. You can set **_fmode** to **O_BINARY** by linking with **BINMODE.OBJ** or by assigning it the value **O_BINARY**. See Section 2.9, "Binary and Text Modes," for a discussion of file-translation modes and the use of the **_fmode** variable.

3.6 _osmajor, _osminor, _osversion

unsigned char _osmajor;
unsigned char _osminor;
unsigned _osversion

The **_osmajor** and **_osminor** variables specify the version number of MS-DOS currently in use. They are declared in **stdlib.h**. The **_osversion** variable provides the complete version number. It is declared in **dos.h**. The **_osmajor** variable holds the "major" version number and the **_osminor** variable stores the "minor" version number. For example, under MS-DOS Version 3.20, **_osmajor** is 3 and **_osminor** is 20.

These variables are useful when you want a program to run on different versions of MS-DOS. For example, you can test the **_osmajor** variable before making a call to **sopen**; if the major version number is earlier (less) than 3, **open** should be used instead of **sopen**.

3.7 environ, _psp

char *environ[ ];
unsigned int _psp;

The **environ** and **_psp** variables provide access to memory areas containing process-specific information. Both variables are declared in the include file **stdlib.h**.

The **environ** variable is an array of pointers to the strings that constitute the process environment. The environment consists of one or more entries of the form

name= string

where *name* is the name of an environment variable and *string* is the value of that variable. The string may be empty. The initial environment settings are taken from the MS-DOS environment at the time of program execution.

The **getenv** and **putenv** routines use the **environ** variable to access and modify the environment table. When **putenv** is called to add or delete environment settings, the environment table changes size, and its location in memory may also change, depending on the program's memory requirements. The **environ** variable is adjusted in these cases and will always point to the correct table location.

The **_psp** variable contains the segment address of the program segment prefix (PSP) for the process. The PSP contains execution information about the process, such as a copy of the command line that invoked the process and the return address on process termination or interrupt. The **_psp** variable can be used to form a long pointer to the PSP, where **_psp** is the segment value and 0 is the offset value.

3.8 Standard Types

A number of run-time library routines use values whose types are defined in include files. These types are listed and described as follows, and the include file that defines each type is given. For a list of the actual type definitions, see the description of the appropriate include file in Chapter 5, "Include Files."

Standard Type	Description
clock_t	The **clock_t** type, defined in **time.h**, stores time values and is used by the **clock** function.
complex	The **complex** structure, defined in **math.h**, stores the real and imaginary parts of complex numbers and is used by the **cabs** function.
diskfree_t	The **diskfree_t** structure, defined in **dos.h**, stores disk information used by the **_dos_getdiskfree** routine.
diskinfo_t	The **diskinfo_t** structure, defined in **bios.h**, records information about disk drives returned by the **_bios_disk** routine.
div_t, ldiv_t	The **div_t** and **ldiv_t** structures, defined in **stdlib.h**, store the values returned by the **div** and **ldiv** functions, respectively.
dosdate_t	The **dosdate_t** structure, defined in **dos.h**, records the current system date used in the **_dos_getdate** and **_dos_setdate** routines.
dostime_t	The **dostime_t** structure, defined in **dos.h**, records the current system time used in the **_dos_gettime** and **_dos_settime** routines.
DOSERROR	The **DOSERROR** structure, defined in **dos.h**, stores values returned by the MS-DOS system call 59H (available under MS-DOS Versions 3.0 and later).

exception	The **exception** structure, defined in **math.h**, stores error information for math routines and is used by the **matherr** routine.
FILE	The **FILE** structure, defined in **stdio.h**, is the structure used in all stream input and output operations. The fields of the **FILE** structure store information about the current state of the stream.
find_t	The **find_t** structure, defined in **dos.h**, stores file-attribute information returned by the **_dos_findfirst** and **_dos_findnext** routines.
fpos_t	The **fgetpos** and **fsetpos** functions use the **fpos_t** object type, defined in **stdio.h**, to record all the information necessary to uniquely specify every position within the file.
jmp_buf	The **jmp_buf** type, defined in **setjmp.h**, is an array type rather than a structure type. It defines the buffer used by the **setjmp** and **longjmp** routines to save and restore the program environment.
onexit_t	The **onexit** routine is declared as an **onexit_t** pointer type, which is defined in **stdlib.h**.
rccoord	The **rccoord** structure, defined in **graph.h**, is used in the graphics library to store the row and column coordinates of the current text output position in the display.
REGS	The **REGS** union, defined in **dos.h**, stores byte and word register values to be passed to and returned from calls to the MS-DOS interface functions.
size_t	The **size_t** type, defined in **stddef.h** and several other include files, is the unsigned integral result of the **sizeof** operator.
sig_atomic_t	The **sig_atomic_t** type, defined in **signal.h**, is the integral type of an object that can be modified as an atomic entity, even in the presence of asynchronous interrupts. It is used in conjunction with the **signal** routine.
SREGS	The **SREGS** structure, defined in **dos.h**, stores the values of the **ES**, **CS**, **SS**, and **DS** registers. This structure is used by the MS-DOS interface functions that require segment register values (**int86x**, **intdosx**, and **segread**).

stat	The **stat** structure, defined in **sys\stat.h**, contains file-status information returned by the **stat** and **fstat** routines.
time_t	The **time_t** type, defined in **time.h**, represents time values in the **mktime** and **time** routines.
timeb	The **timeb** structure, defined in **sys\timeb.h**, is used by the **ftime** routine to store the current system time in a broken-down format.
tm	The **tm** structure, defined in **time.h**, is used by the **asctime**, **gmtime**, and **localtime** functions to store and retrieve time information.
utimbuf	The **utimbuf** structure, defined in **sys\utime.h**, stores file access and modification times used by the **utime** function to change file-modification dates.
va_list	The **va_list** array type, defined in **stdarg.h**, is used to hold information needed by the **va_arg** macro and the **va_end** routine. The called function declares a variable of type **va_list**, which may be passed as an argument to another function.
videoconfig	The **videoconfig** graphics-library structure is defined in **graph.h**. It stores configuration information about the hardware graphics environment.
xycoord	The **xycoord** structure, defined in **graph.h**, is used in the graphics library to store pixel coordinates.

CHAPTER 4

RUN-TIME ROUTINES BY CATEGORY

4.1 Introduction

This chapter describes the major categories of routines included in the C run-time libraries. The discussions of these categories are intended to give a brief overview of the capabilities of the run-time library. For a complete description of the syntax and use of each routine, see Part 2, "Reference."

4.2 Buffer Manipulation

Routine	Use
memccpy	Copies characters from one buffer to another, until a given character or a given number of characters has been copied
memchr	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer
memcmp	Compares a specified number of characters from two buffers
memicmp	Compares a specified number of characters from two buffers without regard to the case of the letters (uppercase and lowercase treated as equivalent)
memmove	Copies a specified number of characters from one buffer to another
memcpy	Copies a specified number of characters from one buffer to another
memset	Uses a given character to initialize a specified number of bytes in the buffer
movedata	Copies a specified number of characters from one buffer to another, even when buffers are in different segments

The buffer-manipulation routines are useful for working with areas of memory on a character-by-character basis. Buffers are arrays of characters (bytes). However, unlike strings, they are not usually terminated with a null character (**'\0'**). Therefore, the buffer-manipulation routines always take a length or count argument.

When the source and target areas overlap, only the **memmove** function is guaranteed to properly copy the full source.

Function declarations for the buffer-manipulation routines are given in the include files **memory.h** and **string.h**.

4.3 Character Classification and Conversion

Routine	Use
isalnum	Tests for alphanumeric character
isalpha	Tests for alphabetic character
isascii	Tests for ASCII character
iscntrl	Tests for control character
isdigit	Tests for decimal digit
isgraph	Tests for printable character except space
islower	Tests for lowercase character
isprint	Tests for printable character
ispunct	Tests for punctuation character
isspace	Tests for white-space character
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal digit
toascii	Converts character to ASCII code
tolower	Tests character and converts to lowercase if uppercase
_tolower	Converts character to lowercase (unconditional)
toupper	Tests character and converts to uppercase if lowercase
_toupper	Converts character to uppercase (unconditional)

The character classification and conversion routines let you test individual characters in a variety of ways and convert between uppercase and lowercase characters. The classification routines identify characters by finding them in a table of classification codes; using these routines to classify characters is generally faster than writing a test expression such as the following:

```
if ((c >= 0) || c <= 0x7f))
```

The **tolower** and **toupper** routines are implemented both as functions and as macros; the remainder of the routines in this category are implemented only as macros. All of the macros are defined in **ctype.h**; this file must be included to use these macros.

The **tolower** and **toupper** macros evaluate their argument twice and therefore cause arguments with side effects to give incorrect results. For this reason, you may want to use the function versions of these routines instead.

The macro versions of **tolower** and **toupper** are used by default when you include **ctype.h**. To use the function versions instead, you must give **#undef** preprocessor directives for **tolower** and **toupper** *after* the **#include** directive for **ctype.h** but *before* you call the routines. This procedure removes the macro definitions and causes occurrences of **tolower** and **toupper** to be treated as function calls to the **tolower** and **toupper** library functions.

If you want to use the function versions of **toupper** and **tolower** and you do not use any of the other character-classification macros in your program, you can simply omit the **ctype.h** include file. In this case no macro definitions are present for **tolower** and **toupper**, so the function versions are used.

Function declarations for the **tolower** and **toupper** functions are given in the include file **stdlib.h** instead of **ctype.h** to avoid conflict with the macro definitions. When you want to use **tolower** and **toupper** as functions and include the declarations from **stdlib.h**, you must follow this sequence:

1. Include **ctype.h** if it is required for other macro definitions.
2. If you include **ctype.h**, give **#undef** directives for **tolower** and **toupper**.
3. Include **stdlib.h**.

The declarations of **tolower** and **toupper** in **stdlib.h** are enclosed in an **#ifndef** block and are processed only if the corresponding identifier (**toupper** or **tolower**) is not defined.

4.4 Data Conversion

Routine	Use
atof	Converts string to **float**
atoi	Converts string to **int**
atol	Converts string to **long**
ecvt	Converts **double** to string
fcvt	Converts **double** to string
gcvt	Converts **double** to string
itoa	Converts **int** to string
ltoa	Converts **long** to string
strtod	Converts string to **double**
strtol	Converts string to a **long** integer
strtoul	Converts string to an **unsigned long** integer
ultoa	Converts **unsigned long** to string

The data-conversion routines convert numbers to strings of ASCII characters and vice versa. These routines are implemented as functions; all are declared in the include file **stdlib.h**. The **atof** function, which converts a string to a floating-point value, is also declared in **math.h**.

4.5 Directory Control

Routine	Use
chdir	Changes current working directory
getcwd	Gets current working directory
mkdir	Makes a new directory
rmdir	Removes a directory

The directory-control routines let you access, modify, and obtain information about the directory structure from within your program. With them, you can get the current working directory, change directories, and add or remove directories.

The directory routines are functions and are declared in the include file **direct.h**.

4.6 File Handling

Routine	Use
access	Checks file-permission setting
chmod	Changes file-permission setting
chsize	Changes file size
filelength	Checks file length
fstat	Gets file-status information
isatty	Checks for character device
locking	Locks areas of file (available with MS-DOS Versions 3.0 and later)
mktemp	Creates unique file name
remove	Deletes file
rename	Renames file
setmode	Sets file-translation mode
stat	Gets file-status information on named file
umask	Sets default-permission mask
unlink	Deletes file

The file-handling routines work on a file designated by a path name, or by a "file handle": a file-management structure returned by the operating system when a file is created or opened. The file-handling routines modify or give information about the designated file. All of these routines except **fstat** and **stat** are declared in the include file **io.h**. The **fstat** and **stat** functions are declared in **sys\stat.h**. The **remove** and **rename** functions are also declared in **stdio.h**.

The **access**, **chmod**, **remove**, **rename**, **stat**, and **unlink** routines operate on files specified by a path name or file name.

The **chsize**, **filelength**, **isatty**, **locking**, **setmode**, and **fstat** routines work with files designated by a file handle.

The **locking** routine works only under MS-DOS Versions 3.0 and later.

The **mktemp** and **umask** routines have slightly different functions than the above routines. The **mktemp** routine creates a unique file name. Programs can use **mktemp** to create unique file names that do not conflict with the names of existing files. The **umask** routine sets the default permission mask for any new files created in a program. The mask may override the permission setting given in the **open** or **creat** call for the new file.

4.7 Graphics

The Microsoft C run-time library includes a graphics library which can be called from Microsoft C as well as from other Microsoft languages that support the C calling conventions. The graphics package supports the IBM (and compatible) Enhanced Graphics Adapter (EGA), Color Graphics Adapter (CGA), and certain operating modes of the Video Graphics Array (VGA) hardware configurations.

4.7.1 Using Graphics Functions

The graphics routines are a large-model library that must be explicitly linked. All graphics functions are declared in the include file **graph.h**. The library can be divided into the seven categories listed below, corresponding to the different tasks involved with creating and manipulating graphic objects:

Task	Description
Configure	Selects the proper display mode for the hardware and establishes memory areas for writing and displaying images
Set coordinates	Specifies the logical origin and the active display area within the screen
Set palette	Specifies a palette mapping
Set attributes	Specifies background and foreground colors and mask and line styles
Output images	Draws and fills figures on the screen
Output text	Writes text to the screen
Transfer images	Stores images in memory and retrieves them

The following sections explain each of these tasks.

4.7.2 Configure

Routine	Use
_displaycursor	Determines whether the cursor will be left on or off on exit from graphics routines
_getvideoconfig	Obtains status of current graphics environment
_setactivepage	Sets memory area for writing images
_setvideomode	Selects screen display mode
_setvisualpage	Sets memory area for displaying images

The **_displaycursor** routine determines whether or not the cursor will be restored on exit from graphics routines. The setting of this routine remains in effect until the routine is called again to change it.

The **_setvideomode** function selects an operating mode for the display hardware.

The **_setactivepage** and **_setvisualpage** functions define memory regions for storing the working page and the displayed page, respectively, in configurations which support multiple video pages.

The **_getvideoconfig** function returns a structure containing information about the hardware environment. Several of the other graphics routines use this information.

4.7.3 Set Coordinates

Routine	Use
_getlogcoord	Converts physical coordinates to logical coordinates
_getphyscoord	Converts logical coordinates to physical coordinates
_setcliprgn	Limits graphic output to part of the screen
_setlogorg	Positions the logical origin
_setviewport	Limits graphic output and positions the logical origin within a limited area

The Microsoft C graphics functions recognize two sets of coordinates:

1. Fixed physical coordinates determined by the hardware and display configuration of the user's environment
2. Logical coordinates defined by the application

The functions in this category alter the logical coordinate system and translate logical coordinates to physical coordinates and vice versa.

The default logical coordinate system is identical to the physical one. The physical origin (0, 0) is always in the upper-left corner of the display. The *x* axis extends in the positive direction left to right, while the *y* axis extends in the positive direction top to bottom. These characteristics are shown in Figure 4.1.

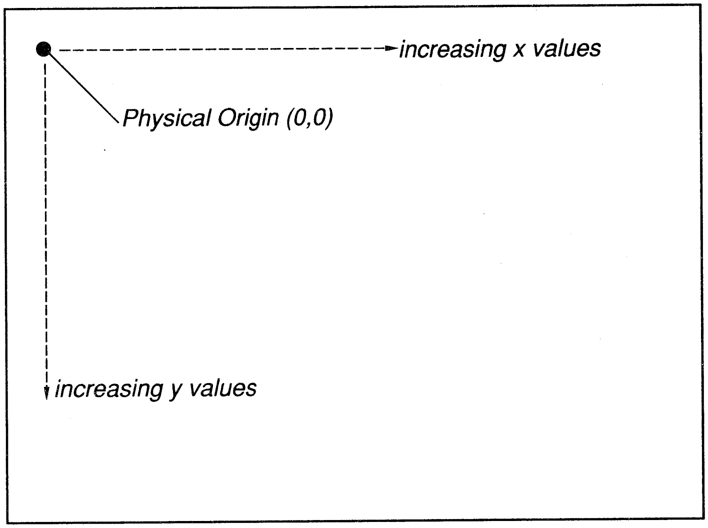


Figure 4.1 The Physical Screen

The dimensions of the x and y axes depend upon the hardware display configuration and the selected mode. These values are accessible at run time by examining the **numxpixels** and **numypixels** fields of the **videoconfig** structure returned by **_getvideoconfig**.

The origin can be moved to a new position relative to the physical origin with the **_setlogorg** function. This function also remaps the pixel coordinates with new logical coordinates, as shown in Figure 4.2.

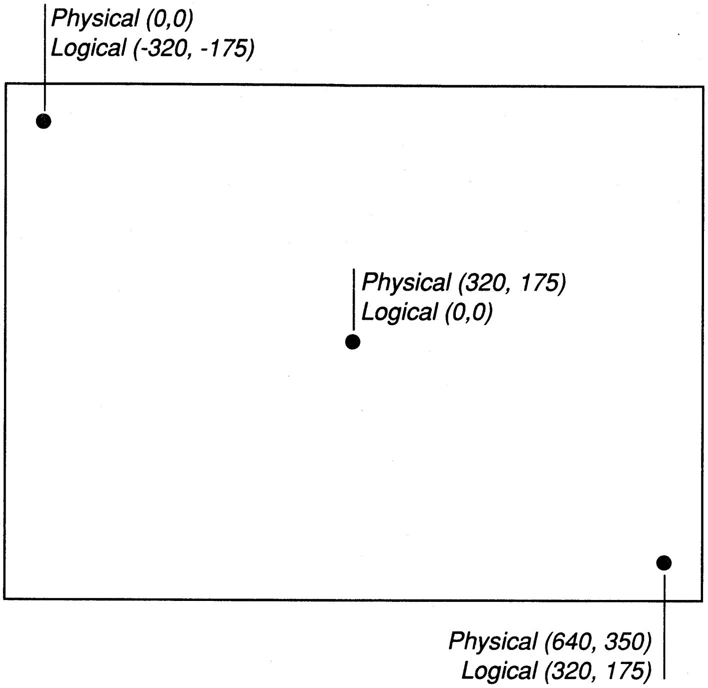


Figure 4.2 The Logical Screen

The physical coordinates of any logical point can be determined with the **_getphyscoord** function, and the logical coordinates of any physical point can be determined with the **_getlogcoord** function.

The **_setcliprgn** function defines a restricted active display area on the screen. The **_setviewport** function does the same thing and also resets the logical origin to the upper-left corner of the restricted active display area.

There is no scaling or built-in axis translation. However, you can do axis translation by redefining the provided interface using C macros to flip the signs of the coordinates.

4.7.4 Set Palette

Routine	Use
_remapallpalette	Assigns colors to all pixel values
_remappalette	Assigns colors to selected pixel values
_selectpalette	Selects a predefined palette

In a graphics mode, a screen pixel can be represented as a one-, two-, or four-bit value, depending upon the particular mode. The representation is called the "pixel value." The range of pixel values can be derived from the **bitsperpixel** field in the **videoconfig** structure returned by **_getvideoconfig**.

Each color that can be displayed is represented by a unique ordinal value. To bind a color ordinal with each pixel value, the graphics library uses the concept of a "palette." A palette is simply a mapping of the actual display colors to the legal pixel values.

Most video modes support only one palette, but the medium-resolution graphics modes, **_MRES4COLOR** and **_MRESNOCOLOR**, support a number of palettes. In these modes, the palette consists of a background color and three other colors. The **_selectpalette** function selects a palette from among the available palettes. All functions that require a color parameter expect to be passed a pixel value.

In addition, the EGA hardware provides the capability of remapping a palette, allowing any available color to be mapped to any pixel value. Two graphics functions allow the EGA configuration to provide this capability: the **_remappalette** function remaps one pixel value; the **_remapallpalette** function remaps the entire palette. These two functions are the only ones that recognize actual color ordinals defined by the display adapter.

Many graphics functions operate only under certain hardware configurations or in certain graphics modes. These functions return a negative value if they are called in an invalid hardware environment.

4.7.5 Set Attributes

Routine	Use
_getbkcolor	Reports the current background color
_getcolor	Obtains the current color
_getlinestyle	Obtains the current line style
_getfillmask	Obtains the current fill mask
_setbkcolor	Sets the current background color
_setcolor	Sets the current color
_setfillmask	Sets the current fill mask
_setlinestyle	Sets the current line style

The output functions (described in Section 4.7.6, "Ouput Images") do not specify color or line-style information. Instead, they rely on a set of current "attributes" which are set independently by the functions listed above.

The **_getcolor** and **_setcolor** functions deal with the "current color" attribute, which is used by the **_floodfill** function, as well as the closed-figure output functions. Similarly, the **_getbkcolor** and **_setbkcolor** functions deal with the "current background color" attribute, employed by the **_clearscreen** function.

The **_getfillmask** and **_setfillmask** functions pertain to the "current fill mask" attribute. The mask is an 8-by-8-bit template array, with each bit representing a pixel. If a bit is 0, the pixel in memory is left untouched: the mask is transparent to that pixel. If a bit is 1, the pixel is assigned the current color value. The template is repeated over the entire fill area.

The **_getlinestyle** and **_setlinestyle** functions pertain to the "current line style" attribute. The line style is determined by a 16-bit template buffer, with each bit corresponding to a pixel. If a bit is 0, the pixel is set to the current background color. If a bit is 1, the pixel is set to the current color. The template is repeated for the length of the line.

4.7.6 Output Images

Routine	Use
_arc	Draws an arc
_clearscreen	Erases the screen and fills it with the current background color
_ellipse	Draws an ellipse
_floodfill	Fills an area of the screen with the current color
_getcurrentposition	Obtains the logical coordinates of the current graphic-output position
_getpixel	Obtains a pixel's value
_lineto	Draws a line from the current graphic output position to a specified point
_moveto	Moves the current graphic-output position to a specified point
_pie	Draws a pie-slice-shaped figure
_rectangle	Draws a rectangle
_setpixel	Sets a pixel's value

These functions assume the presence of current line-style, fill-mask, background-color, and foreground-color attributes to specify their associated parameters. You must write separate calls to select a particular line style, mask, background color, or foreground color. Subsequent output routines employ these parameters.

Circular figures, such as arcs and ellipses, are centered within a "bounding rectangle," specified by two points that define the diagonally opposed corners of the rectangle. The center of the rectangle becomes the center of the figure, and the rectangle's borders determine the size of the figure. Figure 4.3 shows start and end vectors and a bounding rectangle.

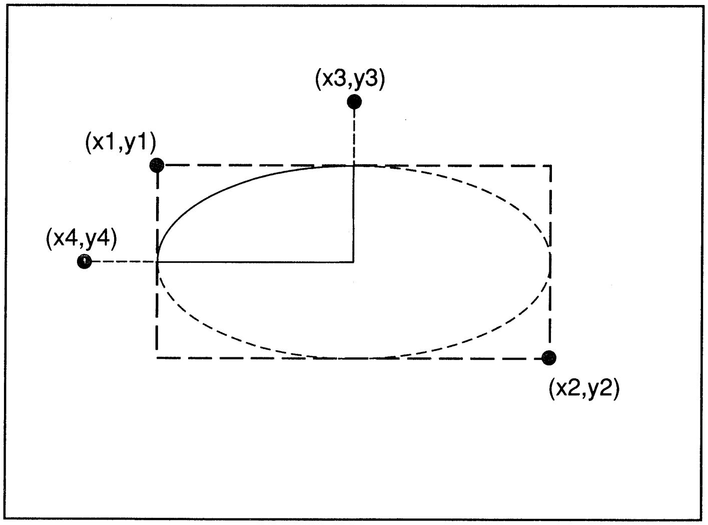


Figure 4.3 Bounding Rectangle

4.7.7 Output Text

Routine	Use
_displaycursor	Sets the cursor "on" or "off" on exit from a graphics routine
_gettextcolor	Obtains the current text color
_gettextposition	Obtains the current text-output position
_outtext	Outputs text to the screen at the current position

_settextposition	Relocates the current text position
_settextcolor	Sets the current text color
_settextwindow	Sets the current text-display window
_wrapon	Enables or disables line wrap

These routines provide text output in both graphics and text modes. Unlike the standard console I/O library routines, these functions recognize window boundaries and should be used in windowing applications.

No formatting capability is provided. If you want to output integer or floating-point values, you must convert the values into a string variable before calling these routines.

All screen positions are specified as character-row and -column coordinates.

The **_settextwindow** routine is analogous to the **_setcliprgn** routine, except that it restricts only the text display area for the **_outtext** routine (it doesn't affect the standard console I/O library routines, such as **_printf**). The **_outtext** routine displays a zero-terminated string on the screen. Text has a color attribute you can obtain with **_gettextcolor** and set with **_settextcolor**. There is also a text-position attribute, which is the current character row and column position where the next text character will be output. This attribute can be obtained and set with the **_gettextposition** and **_settextposition** functions, respectively.

The **_wrapon** function turns on or off line-wrapping of text output. "Line wrapping" refers to breaking a line of text and starting a new one when output encounters a window boundary. Without line-wrap, lines are truncated at the window boundary.

4.7.8 Transfer Images

Routine	Use
_getimage	Stores a screen image in memory
_imagesize	Returns image size in bytes
_putimage	Retrieves an image from memory and displays it

These functions transfer screen images between memory and the display, using a buffer allocated by the application. The **_imagesize** function returns the size in bytes of the buffer needed to store a given image.

4.8 Input and Output

The input and output routines of the standard C library allow you to read and write data to and from files and devices. In C, there are no predefined file structures; all data are treated as sequences of bytes. The following three types of input and output (I/O) functions are available:

1. Stream I/O
2. Low-level I/O
3. Console and port I/O

The "stream" functions treat a data file or data item as a stream of individual characters. By choosing among the many stream functions available, you can process data in different sizes and formats, from single characters to large data structures.

When a file is opened for I/O using the stream functions, the opened file is associated with a structure of type **FILE** (defined in **stdio.h**) containing basic information about the file. A pointer to the **FILE** structure is returned when the stream is opened. Subsequent operations use this pointer (also called the "stream pointer," or just "stream") to refer to the file.

The stream functions provide for buffered, formatted, or unformatted input and output. When a stream is buffered, data that is read from or written to the stream is collected in an intermediate storage location called a buffer. In write operations, the output buffer's contents are written to the appropriate final location when the buffer is full, the stream is closed, or the program terminates normally. The buffer is said to be "flushed" when this occurs. In read operations, a block of data is placed in the input buffer and data are read from the buffer; when the input buffer is empty, the next block of data is transferred into the buffer.

Buffering produces efficient I/O because the system can transfer a large block of data in a single operation rather than performing an I/O operation each time a data item is read from or written to a stream. However, if a program terminates abnormally, output buffers may not be flushed, resulting in loss of data.

The console and port I/O routines can be considered an extension of the stream routines. They allow you to read or write to a console (terminal) or an input/output port (such as a printer port). The port I/O routines simply read and write data in bytes. Some additional options are available with console I/O routines. For example, you can detect whether a character has been typed at the console. You can also choose between echoing characters to the screen as they are read, or reading characters without echoing.

The "low-level" input and output routines do not perform buffering and formatting; rather, they invoke the operating system's input and output capabilities directly. These routines let you access files and peripheral devices at a more basic level than the stream functions.

When a file is opened with a low-level routine, a file "handle" is associated with the opened file. This handle is an integer value that is used to refer to the file in subsequent operations.

Warning

> Stream routines and low-level routines are generally incompatible, so either stream or low-level functions should be used consistently on a given file. Since stream functions are buffered and low-level functions are not, attempting to access the same file or device by two different methods causes confusion and may result in the loss of data in buffers.

4.8.1 Stream Routines

Routine	Use
clearerr	Clears the error indicator for a stream
fclose	Closes a stream
fcloseall	Closes all open streams
fdopen	Opens a stream using its handle
feof	Tests for end-of-file on a stream
ferror	Tests for error on a stream
fflush	Flushes a stream
fgetc	Reads a character from a stream (function version)
fgetchar	Reads a character from **stdin** (function version)
fgetpos	Gets the position indicator of a stream
fgets	Reads a string from a stream
fileno	Gets file handle associated with a stream
flushall	Flushes all streams
fopen	Opens a stream

fprintf	Writes formatted data to a stream
fputc	Writes a character to a stream (function version)
fputchar	Writes a character to **stdout** (function version)
fputs	Writes a string to a stream
fread	Reads unformatted data from a stream
freopen	Reassigns a **FILE** pointer
fscanf	Reads formatted data from a stream
fseek	Repositions **FILE** pointer to given location
fsetpos	Sets the position indicator of a stream
ftell	Gets current **FILE** pointer position
fwrite	Writes unformatted data items to a stream
getc	Reads a character from a stream (macro version)
getchar	Reads a character from **stdin** (macro version)
gets	Reads a line from **stdin**
getw	Reads a binary **int** item from stream
printf	Writes formatted data to **stdout**
putc	Writes a character to a stream (macro version)
putchar	Writes a character to **stdout** (macro version)
puts	Writes a line to a stream
putw	Writes a binary **int** item to a stream
rewind	Repositions **FILE** pointer to beginning of a stream
rmtmp	Removes temporary files created by **tmpfile**
scanf	Reads formatted data from **stdin**
setbuf	Controls stream buffering
setvbuf	Controls stream buffering and buffer size
sprintf	Writes formatted data to string
sscanf	Reads formatted data from string
tempnam	Generates a temporary file name in given directory
tmpfile	Creates a temporary file
tmpnam	Generates a temporary file name

ungetc	Places a character in the buffer
vfprintf	Writes formatted data to a stream
vprintf	Writes formatted data to **stdout**
vsprintf	Writes formatted data to a string

To use the stream functions you must include the file **stdio.h** in your program. This file defines constants, types, and structures used in the stream functions, and contains function declarations and macro definitions for the stream routines.

Some of the constants defined in **stdio.h** may be useful in your program. The manifest constant **EOF** is defined to be the value returned at end-of-file. **NULL** is the null pointer. **FILE** is the structure that maintains information about a stream. **BUFSIZ** defines the default size of stream buffers, in bytes.

4.8.1.1 Opening a Stream

A stream must be opened using the **fdopen**, **fopen**, or **freopen** function before input and output can be performed on that stream. When opening a stream, the named stream can be opened for reading, writing, or both, and can be opened either in text or in binary mode.

The **fdopen**, **fopen**, and **freopen** functions return a **FILE** pointer, which is used to refer to the stream. When you call one of these functions, assign the return value to a **FILE** pointer variable and use that variable to refer to the opened stream. For example, if your program contains the line

```
infile = fopen ("test.dat", "r");
```

you can use the **FILE** pointer variable `infile` to refer to the stream.

4.8.1.2 Predefined Stream Pointers: stdin, stdout, stderr, stdaux, stdprn

When a program begins execution, five streams are automatically opened. These streams are the standard input, standard output, standard error, standard auxiliary, and standard print. By default, the standard input, standard output, and standard error refer to the user's console. This means that whenever a program expects input from the "standard input," it receives that input from the console. Similarly, a program that writes to the "standard output" prints its data to the console. Error messages generated by the library routines are sent to the "standard error," meaning that error messages appear on the user's console.

The assignment of the "standard auxiliary" and "standard print" streams depends on the machine configuration; these streams usually refer to an auxiliary port and a printer, respectively, but they might not be set up on a particular system. Be sure to check your machine configuration before using these streams.

You can refer to the five standard system streams by using the following predefined file handles:

Handle	Stream
stdin	Standard input
stdout	Standard output
stderr	Standard error
stdaux	Standard auxiliary
stdprn	Standard print

You can use these pointers in any function that requires a stream pointer as an argument. Some functions, such as **getchar** and **putchar**, are designed to use **stdin** or **stdout** automatically. The pointers **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** are constants, not variables; do not try to assign them a new stream pointer value.

You can use the MS-DOS redirection symbols (<, >, or >>) or the pipe symbol (|) to redefine the standard input and standard output for a particular program. (See your operating-system manual for a complete discussion of redirection and pipes.) For example, if you execute a program and redirect its output to a file named `results`, the program writes to the `results` file each time the standard output is specified in a write operation. Note that you don't change the program when you redirect the output. You simply change the file associated with **stdout** for a single execution of the program.

You can redefine **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** so that it refers to a disk file or to a device. The **freopen** routine is used for this purpose. For a description of this option, see the **freopen** description in the reference section of this manual.

Important

> At the MS-DOS command level, **stderr** (standard error) cannot be redirected.

4.8.1.3 Controlling Stream Buffering

Files opened using the stream functions are buffered by default, except for the preopened streams **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. The **stderr** and **stdaux** streams are unbuffered by default, unless they are used in one of the **printf** or **scanf** family of functions, in which case they are assigned a temporary buffer. These two streams can also be buffered with **setbuf** or **setvbuf**. The **stdin**, **stdout**, and **stdprn** streams are buffered; each buffer is flushed whenever it is full, or whenever the function causing I/O terminates.

By using the **setbuf** or **setvbuf** functions, you can cause a stream to be unbuffered, or you can associate a buffer with an unbuffered stream. Buffers allocated by the system are not accessible to the user, but buffers allocated with **setbuf** or **setvbuf** are named by the user and can be manipulated as if they were variables. Buffers can be any size: if you use **setbuf**, this size is set by the manifest constant **BUFSIZ** in **stdio.h**; if you use **setvbuf**, you can set the size of the buffer yourself. (See **setbuf** and **setvbuf** in the reference section of this manual.)

Buffers are automatically flushed when they are full, when the associated file is closed, or when a program terminates normally. You can flush buffers at other times by using the **fflush** and **flushall** routines. The **fflush** routine flushes a single specified stream, while **flushall** flushes all streams that are open and buffered.

4.8.1.4 Closing Streams

The **fclose** and **fcloseall** functions close a stream or streams. The **fclose** routine closes a single specified stream; **fcloseall** closes all open streams except **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. If your program does not explicitly close a stream, the stream is automatically closed when the program terminates. However, it is a good practice to close a stream when finished with it, as the number of streams that can be open at a given time is limited.

4.8.1.5 Reading and Writing Data

The stream functions allow you to transfer data in a variety of ways. You can read and write binary data (a sequence of bytes), or specify reading and writing by characters, lines, or more complicated formats. A list at the beginning of this section summarizes the stream functions for reading and writing data; for a full description of each function, see the reference section of this manual.

Reading and writing operations on streams always begin at the current position of the stream, known as the "file pointer" for the stream. The file pointer is changed to reflect the new position after a read or write operation takes place. For example, if you read a single character from a stream, the file pointer is increased by one byte so that the next operation begins with the first unread character. If a stream is opened for appending, the file pointer is automatically positioned at the end of the file before each write operation.

The **feof** macro detects an end-of-file condition on a stream. Once the end-of-file indicator is set, it remains set until the file is closed, or until **clearerr** or **rewind** is called.

You can position the file pointer anywhere in a file by using the **fseek** function. The next operation occurs at the position you specified. The **rewind** function positions the file pointer at the beginning of the file. Use the **ftell** function to determine the current position of the file pointer.

Streams associated with a device (such as a console) do not have file pointers. Data coming from or going to a console cannot be accessed randomly. Routines that set or get the file-pointer position (such as **fseek**, **fgetpos**, **fsetpos**, **ftell**, or **rewind**) have undefined results if used on a stream associated with a device.

4.8.1.6 Detecting Errors

When an error occurs in a stream operation, an error indicator for the stream is set. You can use the **ferror** macro to test the error indicator and determine whether an error has occurred. Once an error has occurred, the error indicator for the stream remains set until the stream is closed, or until you explicitly clear the error indicator by calling **clearerr** or **rewind**.

4.8.2 Low-Level Routines

Routine	Use
close	Closes a file
creat	Creates a file
dup	Creates a second handle for a file
dup2	Reassigns a handle to a file
eof	Tests for end-of-file
lseek	Repositions file pointer to a given location

open	Opens a file
read	Reads data from a file
sopen	Opens a file for file sharing
tell	Gets current file-pointer position
write	Writes data to a file

Low-level input and output calls do not buffer or format data. Files opened by low-level calls are referenced by a "file handle," an integer value used by the operating system to refer to the file. The **open** function is used to open files; on MS-DOS Versions 3.0 and later, **sopen** can be used to open a file with file-sharing attributes.

Low-level functions, unlike the stream functions, do not require the include file **stdio.h**. However, some common constants are defined in **stdio.h**; for example, the end-of-file indicator, **EOF**, may be useful. If your program requires these constants, you must include **stdio.h**.

Declarations for the low-level functions are given in the include file **io.h**.

4.8.2.1 Opening a File

A file must be opened with the **open**, **sopen**, or **creat** function before input and output with the low-level functions can be performed on that file. The file can be opened for reading, writing, or both, and opened in either text or binary mode. The include file **fcntl.h** must be included when opening a file, as it contains definitions for flags used in **open**. In some cases the files **sys\types.h** and **sys\stat.h** must also be included; for more information see the reference page for **open** in the reference section of this manual.

These functions return a file handle, to be used to refer to the file in later operations. When you call one of these functions, assign the return value to an integer variable and use that variable to refer to the opened file.

4.8.2.2 Predefined Handles

When a program begins execution, five file handles, corresponding to the standard input, standard output, standard error, standard auxiliary, and standard print, are already assigned. By using the following predefined handles, a program can call low-level functions to access the standard input, standard output, standard error, standard auxiliary, and standard print streams (described with the stream functions in Section 4.8.1.2):

Stream	Handle
stdin	0
stdout	1
stderr	2
stdaux	3
stdprn	4

You can use these file handles in your program without previously opening the associated files. They are automatically opened when the program begins, as shown by the output from the following short program, which uses the **fileno** function to print the file-handle values assigned to the standard input, standard output, standard error, standard auxiliary, and standard print streams:

```
#include <stdio.h>

main( )
        {
        printf("stdin:  %d\n", fileno(stdin));
        printf("stdout: %d\n", fileno(stdout));
        printf("stderr: %d\n", fileno(stderr));
        printf("stdaux: %d\n", fileno(stdaux));
        printf("stdprn: %d\n", fileno(stdprn));
        }
```

Output:

```
stdin:  0
stdout: 1
stderr: 2
stdaux: 3
stdprn: 4
```

As with the stream functions, you can use redirection and pipe symbols when you execute your program to redirect the standard input and standard output. The **dup** and **dup2** functions allow you to assign multiple handles for the same file; these functions are typically used to associate the predefined file handles with different files.

Important

> At the MS-DOS command level, **stderr** (standard error) cannot be redirected.

4.8.2.3 Reading and Writing Data

Two basic functions, **read** and **write**, perform input and output. As with the stream functions, reading and writing operations always begin at the current position in the file. The current position is updated each time a read or write operation occurs.

The **eof** routine can be used to test for an end-of-file condition. Low-level I/O routines set the **errno** variable when an error occurs. This means that you can use the **perror** function to print information about I/O errors, or the **strerror** function to store this error information in a string.

You can position the file pointer anywhere in a file by using the **lseek** function; the next operation occurs at the position you specified. Use the **tell** function to determine the current position of the file pointer.

Devices (such as the console) do not have file pointers. The **lseek** and **tell** routines have undefined results if used on a handle associated with a device.

4.8.2.4 Closing Files

The **close** function closes an open file. Open files are automatically closed when a program terminates. However, it is good practice to close a file when finished with it, as the number of files that can be open at a given time is limited.

4.8.3 Console and Port I/O

Routine	Use
cgets	Reads a string from the console
cprintf	Writes formatted data to the console
cputs	Writes a string to the console
cscanf	Reads formatted data from the console
getch	Reads a character from the console
getche	Reads a character from the console and echoes it
inp	Reads one byte from the specified I/O port
inpw	Reads a two-byte word from the specified I/O port
kbhit	Checks for a keystroke at the console

outp	Writes one byte to the specified I/O port
outpw	Writes a two-byte word to the specified I/O port
putch	Writes a character to the console
ungetch	"Ungets" the last character read from the console so that it becomes the next character read

The console and port I/O routines are implemented as functions and are declared in the include file **conio.h**. These functions perform reading and writing operations on your console or on the specified port. The **cgets**, **cscanf**, **getch**, **getche**, and **kbhit** routines take input from the console, while **cprintf**, **cputs**, **putch**, and **ungetch** write to the console. The input or output of these functions can be redirected. (Redirection occurs at the operating-system level; the library itself has no control over it.)

The console or port does not have to be opened or closed before I/O is performed, so there are no open or close routines in this category. The port I/O routines **inp** and **outp** read or write one byte at a time from the specified port. The routines **inpw** and **outpw** read or write two-byte words, respectively.

The console I/O routines allow reading and writing of strings (**cgets** and **cputs**), formatted data (**cscanf** and **cprintf**), and characters. Several options are available when reading and writing characters.

The **putch** routine writes a single character to the console. The **getch** and **getche** routines read a single character from the console; **getche** echoes the character back to the console, while **getch** does not. The **ungetch** routine "ungets" the last character read; the next read operation on the console begins with the "ungotten" character.

The **kbhit** routine determines whether a key has been struck at the console. This routine allows you to test for keyboard input before you attempt to read from the console.

Notes

The console I/O routines use the corresponding MS-DOS system calls to read and write characters. Since these routines are not compatible with stream or low-level library routines, console routines should not be used with them.

4.9 Math

Routine	Use
acos	Calculates arc cosine
asin	Calculates arc sine
atan	Calculates arc tangent
atan2	Calculates arc tangent
bessel[1]	Calculates Bessel functions
cabs	Finds absolute value of a complex number
ceil	Finds integer ceiling
_clear87[2]	Gets and clears floating-point status word
_control87[2]	Gets old floating-point control word and sets new control-word value
cos	Calculates cosine
cosh	Calculates hyperbolic cosine
dieeetomsbin	Converts IEEE double-precision number to Microsoft binary format
dmsbintoieee	Converts Microsoft binary double-precision number to IEEE format
exp	Calculates exponential function
fabs	Finds absolute value
fieeetomsbin	Converts IEEE single-precision number to Microsoft binary format
floor	Finds largest integer less than or equal to argument
fmod	Finds floating-point remainder
fmsbintoieee	Converts Microsoft binary single-precision number to IEEE format
_fpreset	Reinitializes the floating-point-math package
frexp	Calculates an exponential value

hypot	Calculates hypotenuse of right triangle
ldexp	Calculates argument times 2^{exp}
log	Calculates natural logarithm
log10	Calculates base-10 logarithm
matherr	Handles math errors
modf	Breaks down argument into integer and fractional parts
pow	Calculates a value raised to a power
sin	Calculates sine
sinh	Calculates hyperbolic sine
sqrt	Finds square root
_status87[2]	Gets the floating-point status word
tan	Calculates tangent
tanh	Calculates hyperbolic tangent

The math routines allow you to perform common mathematical calculations. All math routines work with floating-point values and therefore require floating-point support (see Section 2.11, "Floating-Point Support"). Function declarations for the math routines are given in the include file **math.h**, with the exception of **_clear87**, **_control87**, **_fpreset**, and **_status87**, whose definitions are given in the **float.h** include file.

The **matherr** routine is invoked by the math functions when errors occur. This routine is defined in the library, but can be redefined by the user if different error-handling procedures are desired. The user-defined **matherr** function, if given, must conform to the specifications given on the **matherr** reference page in Part 2 of this manual.

You are not required to supply a definition for **matherr**. If no definition is present, the default error returns for each routine are used. See the reference page for each routine in Part 2 of this manual for a description of that routine's error returns.

---

[1] The **bessel** routine does not correspond to a single function, but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

[2] Not available with the **/FPa** compiler option

4.10 Memory Allocation

Routine	Use
alloca	Allocates a block of memory from the program's stack
calloc	Allocates storage for array
_expand	Reallocates block of memory without moving its location
_ffree	Frees a block allocated by **_fmalloc**
_fheapchk	Checks the memory space outside the default data segment (far heap) for consistency
_fheapset	Fills the free far heap entries with a specified value
_fheapwalk	Walks through the far heap, one entry at a time, and returns information about each far heap entry
_fmalloc	Allocates a block of memory outside the far heap and returns a far pointer
free	Frees a block allocated with **calloc**, **malloc**, or **realloc**
_freect	Returns approximate number of items of given size that could be allocated
_fmsize	Returns size of memory block pointed to by far pointer
halloc	Allocates storage for huge array
_heapchk	Checks the heap for consistency
_heapset	Fills the free heap entries with a specified value
_heapwalk	Walks through the heap, one entry at a time, and returns information about each heap entry
hfree	Frees a block allocated by **halloc**
malloc	Allocates a block
_memavl	Returns approximate number of bytes available in memory for allocation
_memmax	Returns size of largest contiguous free space in the near heap
_msize	Returns size of block allocated by **calloc**, **malloc**, or **realloc**

_nfree	Frees a block allocated by **_nmalloc**
_nheapchk	Checks the near heap (default data segment) for consistency
_nheapset	Fills the free near heap entries with a specified value
_nheapwalk	Walks through the near heap, one entry at a time, and returns information about each near heap entry
_nmalloc	Allocates a block of memory in default data segment, returns a near pointer
_nmsize	Returns size of memory block pointed to by near pointer
realloc	Reallocates a block
sbrk	Resets break value
stackavail	Returns size of stack space available for allocation with **alloca**

The memory-allocation routines allow you to allocate, free, and reallocate blocks of memory. They are declared in the include file **malloc.h**.

When a program written in Microsoft C is loaded for execution, it first shrinks its MS-DOS-allocated memory to fit within a single 64K data segment. This is true even though the program header indicates that all of memory is allocated for the program. The extent to which the program's memory allocation is reduced can be altered with the **/CPARMAXALLOC** linker option, described in your compiler guide.

The **calloc** and **malloc** routines allocate memory blocks. The **malloc** routine allocates a given number of bytes, while **calloc** allocates and initializes to 0 an array with elements of a given size. In small data models (small- and medium-model programs), **malloc** maps to (is defined as) **_nmalloc**, and **free** maps to **_nfree**. In large data models (compact- and large-model programs), **malloc** maps to **_fmalloc**, and **free** maps to **_ffree**.

The **halloc** routine performs essentially the same function as **calloc**, with the difference that **halloc** can be used to allocate space for huge arrays (those exceeding 64K in size). Arrays greater than 64K allocated with **halloc** must satisfy the requirements for huge arrays discussed in your compiler user's guide.

When **_nmalloc** is called, it allocates from the default data segment ("near heap"), and **_nfree** releases memory back to the near heap. The first time **_fmalloc** is called, it allocates an additional segment from MS-DOS, then returns to the calling program a pointer to the requested amount of memory. It performs heap management on the rest of the segment for subsequent calls until that segment has been completely allocated, then gets another segment from MS-DOS, and so on. The **_ffree** function returns allocated memory to the heap block it came from, without releasing it back to MS-DOS. If **_fmalloc** runs out of MS-DOS memory to allocate, it will attempt to allocate from the near heap as a last resort.

The **halloc** and **hfree** routines differ from **_nmalloc**/**_nfree** and **_fmalloc**/**_nfree** in that the **halloc** and **hfree** allocate and free memory directly from MS-DOS, instead of in the near or far heap space. The **halloc** function does not do heap management on the MS-DOS memory space. When **hfree** is called, it simply returns memory back to MS-DOS.

The **_nmalloc** function is fastest and should be used in small-model programs where total memory allocation is less than 64K. The exact amount of memory available for near heap allocations depends on how much of the default data segment is used by the stack, program data, and run-time data. The **_fmalloc** function is slower. It should be used when total memory allocation requirements are too large to use **_nmalloc**, but no single data object is greater than 64K.

The **halloc** function is the slowest of all because it allocates from MS-DOS for every request; however, it is useful in two cases:

- When you want data objects larger than 64K
- When you want to make sure you can free allocated memory back to MS-DOS for subsequent calls to the **spawn** functions

The **realloc** and **_expand** routines change the size of an allocated block. The **_expand** function always attempts to change the size of an allocated block without moving its heap location; it expands the size of the block to the size requested, or as much as the current location will allow, whichever is smaller. In contrast, **realloc** changes the location in the heap if there is not enough room.

The **halloc** routine returns a huge pointer to a **char** item, **_fmalloc** returns a far pointer to a **char** item, and **_nmalloc** returns a near pointer to a **char** item; all other allocation routines return a **char** pointer. The spaces to which these routines point satisfy the alignment requirements for any type of object. When allocating items of types other than **char**, use a type cast on the return value.

The **_freect** and **_memavl** routines tell you how much memory is available for dynamic memory allocation in the default data segment. The

_freect routine returns the approximate number of items of a given size that can be allocated, while **_memavl** returns the total number of bytes available for allocation requests.

The **_msize** function returns the size of a memory block allocated by a call to **calloc**, **_expand**, **malloc**, or **realloc**. The functions **_fmsize** and **_nmsize** return the size of a memory block allocated by a call to **_fmalloc** or **_nmalloc**, respectively.

The **sbrk** routine is a lower-level memory-allocation routine. It increases the program's break value (the address of the first location beyond the end of the default data segment), allowing the program to take advantage of available unallocated memory.

Warning

> In general, a program that uses the **sbrk** routine should not use the other memory-allocation routines, although their use is not prohibited. In particular, using **sbrk** to decrease the break value may cause unpredictable results from subsequent calls to the other memory-allocation routines.

The preceding routines all allocate memory dynamically from the heap. Microsoft C also provides two memory functions, **alloca** and **stackavail**, for allocating space from the stack and determining the amount of available stack space. The **alloca** routine allocates the requested number of bytes from the stack, which are freed when control returns from the function calling **alloca**. The **stackavail** routine lets your program know how much memory (in bytes) is available on the stack.

4.11 Process Control

Routine	Use
abort	Aborts a process
atexit	Executes functions at program termination
execl	Executes child process with argument list
execle	Executes child process with argument list and given environment

execlp	Executes child process using **PATH** variable and argument list
execlpe	Executes child process using **PATH** variable, given environment, and argument list
execv	Executes child process with argument array
execve	Executes child process with argument array and given environment
execvp	Executes child process using **PATH** variable and argument array
execvpe	Executes child process using **PATH** variable, given environment, and argument array
exit	Terminates process
_exit	Terminates process without flushing buffers
getpid	Gets process ID number
onexit	Executes functions at program termination
raise	Sends a signal to the calling process
signal	Handles an interrupt signal
spawnl	Executes child process with argument list
spawnle	Executes child process with argument list and given environment
spawnlp	Executes child process using **PATH** variable and argument list
spawnlpe	Executes child process using **PATH** variable, given environment, and argument list
spawnv	Executes child process with argument array
spawnve	Executes child process with argument array and given environment
spawnvp	Executes child process using **PATH** variable and argument array
spawnvpe	Executes child process using **PATH** variable, given environment, and argument array
system	Executes an MS-DOS command

The term "process" refers to a program being executed by the operating system. A process consists of the program's code and data, plus information pertaining to the status of the process, such as the number of open files. Whenever you execute a program at the MS-DOS level, you start a process. In addition, you can start, stop, and manage processes from within a program by using the process-control routines.

The process-control routines allow you to do the following:

1. Identify a process by a unique number (**getpid**)
2. Terminate a process (**abort**, **exit**, and **_exit**)
3. Call a new function when a process terminates (**atexit**, **onexit**)
4. Handle an interrupt signal (**signal**)
5. Send a signal to a process (**raise**)
6. Start a new process (the **exec** and **spawn** families of routines, plus the **system** routine)

All process-control functions except **signal** are declared in the include file **process.h**. The **signal** function is declared in **signal.h**. The **abort**, **exit**, and **system** functions are also declared in the **stdlib.h** include file.

The **abort** and **_exit** functions perform an immediate exit without flushing stream buffers. The **exit** call performs an exit after flushing stream buffers.

The **atexit** and **onexit** functions both create a list of functions to be executed when the calling program exits; the only difference between the two is that **atexit** is part of the draft proposed ANSI standard. The **onexit** function is retained for compatibility with previous versions of Microsoft C.

The **system** call executes a given MS-DOS command. The **exec** and **spawn** routines start a new process, called the "child" process. The difference between the **exec** and **spawn** routines is that the **spawn** routines are capable of returning control from the child process to its caller (the "parent" process). Both the parent process and the child process are present in memory (unless **P_OVERLAY** is specified).

In the **exec** routines, the child process overlays the parent process, so returning control to the parent process is impossible (unless an error occurs when attempting to start execution of the child process).

There are eight forms each of the **spawn** and **exec** routines. The differences between the forms are summarized in Table 4.1. The function names are given in the first column. The second column specifies whether the current **PATH** setting is used to locate the file to be executed as the child process.

The third column describes the method for passing arguments to the child process. Passing an argument list means that the arguments to the child process are listed as separate arguments in the **exec** or **spawn** call; passing an argument array means that the arguments are stored in an array, and a pointer to the array is passed to the child process. The argument-list method is typically used when the number of arguments is constant or is known at compile time, while the argument-array method is useful when the number of arguments must be determined at run time.

The last column specifies whether the child process inherits the environment settings of its parent or whether a table of environment settings can be passed to set up a different environment for the child process.

Table 4.1

Forms of the spawn and exec Routines

Routines	Use of PATH Setting	Argument-Passing Convention	Environment
execl, **spawnl**	Do not use **PATH**	Argument list	Inherited from parent
execle, **spawnle**	Do not use **PATH**	Argument list	Pointer to environment table for child process passed as last argument
execlp, **spawnlp**	Use **PATH**	Argument list	Inherited from parent
execlpe, **spawnlpe**	Use **PATH**	Argument list	Pointer to environment table for child process passed as last argument
execv, **spawnv**	Do not use **PATH**	Argument array	Inherited from parent
execve, **spawnve**	Do not use **PATH**	Argument array	Pointer to environment table for child process passed as last argument
execvp, **spawnvp**	Use **PATH**	Argument array	Inherited from parent
execvpe, **spawnvpe**	Use **PATH**	Argument array	Pointer to environment table for child process passed as last argument

4.12 Searching and Sorting

Routine	Use
bsearch	Performs binary search
lfind	Performs linear search for given value
lsearch	Performs linear search for given value, which is added to array if not found
qsort	Performs quick sort

The **bsearch**, **lfind**, **lsearch**, and **qsort** functions provide helpful binary-search, linear-search, and quick-sort utilities. They are declared in the include file **search.h**.

4.13 String Manipulation

Routine	Use
strcat	Appends a string
strchr	Finds first occurrence of a given character in string
strcmp	Compares two strings
strcmpi	Compares two strings without regard to case ("i" indicates that this function is case insensitive)
strcpy	Copies one string to another
strcspn	Finds first occurrence of a character from given character set in string
strdup	Duplicates string
strerror	Saves system-error message and optional user-error message in string
stricmp	Compares two strings without regard to case (identical to **strcmpi**)
strlen	Finds length of string
strlwr	Converts string to lowercase
strncat	Appends characters of string

strncmp	Compares characters of two strings
strncpy	Copies characters of one string to another
strnicmp	Compares characters of two strings without regard to case ("i" indicates that this function is case insensitive)
strnset	Sets characters of string to given character
strpbrk	Finds first occurrence of character from one string in another
strrchr	Finds last occurrence of given character in string
strrev	Reverses string
strset	Sets all characters of string to given character
strspn	Finds first substring from given character set in string
strstr	Finds first occurrence of given string in another string
strtok	Finds next token in string
strupr	Converts string to uppercase

The string functions are declared in the include file **string.h**. A wide variety of string functions is available in the run-time library. With these functions, you can do the following:

- Perform string comparisons
- Search for strings, individual characters, or characters from a given set
- Copy strings
- Convert strings to a different case
- Set characters of the string to a given character
- Reverse the characters of strings
- Break strings into tokens
- Store error messages in a string

All string functions work on null-terminated character strings. When working with character arrays that do not end with a null character, you can use the buffer-manipulation routines, described earlier in this chapter.

4.14 System Calls

The following routines give access to BIOS (Basic Input/Output System) interrupts and MS-DOS system calls.

4.14.1 BIOS Interface

Routine	Use
_bios_disk	Issues service requests for both hard and floppy disks, using INT 0x13
_bios_equiplist	Performs an equipment check, using INT 0x11
_bios_keybrd	Provides access to keyboard services, using INT 0x16
_bios_memsize	Obtains information about available memory, using INT 0x12
_bios_printer	Performs printer output services, using INT 0x17
_bios_serialcom	Performs serial communications tasks, using INT 0x14
_bios_timeofday	Provides access to system clock, using INT 0x1A

The functions in this category provide direct access to the BIOS interrupt services. They are all declared in **bios.h**.

4.14.2 MS-DOS Interface

Routine	Use
bdos	Invokes MS-DOS system call; uses only **DX** and **AL** registers
_chain_intr	Chains one interrupt handler to another
_disable	Disables interrupts
_dos_allocmem	Allocates a block of memory, using MS-DOS system call 0x48
_dos_close	Closes a file, using MS-DOS system call 0x3E
_dos_creat	Creates a new file and erases any existing file having the same name, using MS-DOS system call 0x3C

_dos_creatnew	Creates a new file and returns an error if a file having the same name exists, using MS-DOS system call 0x5B
_dos_findfirst	Finds first occurrence of a given file, using MS-DOS system call 0x4E
_dos_findnext	Finds subsequent occurrences of a given file, using MS-DOS system call 0x4F
_dos_freemem	Frees a block of memory, using MS-DOS system call 0x49
_dos_getdate	Gets the system date, using MS-DOS system call 0x2A
_dos_getdiskfree	Gets information on a disk drive, using MS-DOS system call 0x36
_dos_getdrive	Gets the current default drive, using MS-DOS system call 0x19
_dos_getfileattr	Gets current attributes of a file or directory, using MS-DOS system call 0x43
_dos_getftime	Gets the date and time a file was last written, using MS-DOS system call 0x57
_dos_gettime	Gets the current system time, using MS-DOS system call 0x2C
_dos_getvect	Gets the current value of a specified interrupt vector, using MS-DOS system call 0x35
_dos_keep	Installs terminate-and-stay-resident (TSR) programs using MS-DOS system call 0x31
_dos_open	Opens an existing file, using MS-DOS system call 0x3D
_dos_read	Reads a file, using MS-DOS system call 0x3F
_dos_setblock	Changes the size of a previously allocated block, using MS-DOS system call 0x4A
_dos_setdate	Sets the current system date, using MS-DOS system call 0x2B
_dos_setdrive	Sets the default disk drive, using MS-DOS system call 0x0E
_dos_setfileattr	Sets the current attributes of a file, using MS-DOS system call 0x43
_dos_setftime	Sets the date and time that the specified file was last written, using MS-DOS system call 0x57

_dos_settime	Sets the system time, using MS-DOS system call 0x2D
_dos_setvect	Sets the current value of the specified interrupt vector, using MS-DOS system call 0x25
_dos_write	Sends output to a file, using MS-DOS system call 0x40
dosexterr	Obtains register values from MS-DOS system call 0x59
_enable	Enables interrupts
FP_OFF	Returns offset portion of a far pointer
FP_SEG	Returns segment portion of a far pointer
_harderr	Establishes a hardware error handler
_hardresume	Returns to MS-DOS after a hardware error
_hardretn	Returns to the application after a hardware error
int86	Invokes MS-DOS interrupts
int86x	Invokes MS-DOS interrupts with segment register values
intdos	Invokes MS-DOS system call using registers other than **DX** and **AL**
intdosx	Invokes MS-DOS system call using registers other than **DX** and **AL** with segment register values
segread	Returns current values of segment registers

These routines are implemented as functions and declared in **dos.h**.

The **_harderr** routine is used to define a hardware-error interrupt handler. The **_hardresume** and **_hardretn** routines are used within a hardware error handler to define the return from the error.

The **dosexterr** function obtains and stores the register values returned by MS-DOS system call 0x59 (extended error handling). This function is provided for use with MS-DOS versions 3.0 and later.

The **bdos** routine is useful for invoking MS-DOS calls that use either or both of the **DX** (**DH**/**DL**) and **AL** registers for arguments. However, **bdos** should not be used to invoke system calls that return an error code in **AX** if the carry flag is set; since the program cannot detect whether the carry flag is set, it cannot determine whether the value in **AX** is a legitimate value or an error value. In this case, the **intdos** routine should be used instead, since it allows the program to detect whether the carry flag is set. The **intdos** routine can also be used to invoke MS-DOS calls that use registers other than **DX** and **AL**.

The **intdosx** routine is similar to the **intdos** routine, but is used when **ES** is required by the system call, when **DS** must contain a value other than the default data segment (for instance, when a **far** pointer is used), or when making the system call in a large-model program. When calling **intdosx**, give an argument that specifies the segment values to be used in the call.

The **int86** routine can be used to invoke MS-DOS interrupts. The **int86x** routine is similar, but, like the **intdosx** routine, is designed to work with large-model programs and far items, as described in the preceding paragraph for **intdosx**.

The **FP_OFF** and **FP_SEG** routines allow easy access to the segment and offset portions of a far pointer value. **FP_OFF** and **FP_SEG** are implemented as macros and defined in **dos.h**.

The **segread** routine returns the current values of the segment registers. This routine is typically used with the **intdosx** and **int86x** routines to obtain the correct segment values.

The **_chain_int** routine is useful for chaining interrupt handlers together. The **_enable** routine enables interrupts, while the **_disable** routine disables interrupts.

The routines prefixed with **_dos_** are all direct system interfaces that use the system calls noted above. More detailed information on these system calls can be found in the *MS-DOS Programmer's Reference.*

Note

Do not use the MS-DOS interface I/O routines in conjunction with the console, low-level, or stream I/O routines.

4.15 Time

Routine	Use
asctime	Converts time from structure to character string
clock	Returns the elapsed CPU time for a process

ctime	Converts time from long integer to character string
difftime	Computes the difference between two times
ftime	Gets current system time as structure
gmtime	Converts time from integer to structure
localtime	Converts time from integer to structure with local correction
mktime	Converts time to a calendar value
_strdate	Returns the current system date as a string
_strtime	Returns the current system time as a string
time	Gets current system time as long integer
tzset	Sets external time variables from environment time variable
utime	Sets file-modification time

The time functions allow you to obtain the current time, then convert and store it according to your particular needs. The current time is always taken from the system time. The **time** and **ftime** functions return the current time as the number of seconds elapsed since Greenwich mean time, January 1, 1970. This value can be converted, adjusted, and stored in a variety of ways, using the **asctime**, **ctime**, **gmtime**, **localtime**, and **mktime** functions. The **utime** function sets the modification time for a specified file, using either the current time or a time value stored in a structure.

The **clock** function returns the elapsed CPU time for the calling process.

The **ftime** function requires two include files: **sys\types.h** and **sys\timeb.h**. The **ftime** function is declared in **sys\timeb.h**. The **utime** function also requires two include files: **sys\types.h** and **sys\utime.h**. The **utime** function is declared in **sys\utime.h**. The remainder of the time functions are declared in the include file **time.h**.

When you want to use **ftime** or **localtime** to make adjustments for local time, you must define an environment variable named **TZ**. See Section 3.2 on the global variables **daylight**, **timezone**, and **tzname** for a discussion of the **TZ** variable; **TZ** is also described on the **tzset** reference page in Part 2 of this manual.

The **_strdate** and **_strtime** routines return strings containing the current date and time, respectively, in the MS-DOS date and time format rather than in the XENIX-style formats.

4.16 Variable-Length Argument Lists

Routine	Use
va_arg	Retrieves argument from list
va_end	Resets pointer
va_start	Sets pointer to beginning of argument list

The **va_arg**, **va_end**, and **va_start** routines are macros that provide a portable way to access the arguments to a function when the function takes a variable number of arguments. Two versions of the macros are available: the macros defined in the **vararg.h** include file, which are compatible with the UNIX System V definition, and the macros defined in **stdarg.h**, which conform to the proposed ANSI C standard.

For more information on the differences between the two versions and for an explanation of how to use the macros, see their descriptions in the reference section of this manual.

4.17 Miscellaneous

Routine	Use
abs	Finds absolute value of integer
assert	Tests for logic error
div	Divides integers
getenv	Gets value of environment variable
labs	Finds absolute value of long integer
ldiv	Divides long integers
longjmp	Restores a saved stack environment
_lrotl	Shifts a **long int** item to the left
_lrotr	Shifts a **long int** item to the right
_makepath	Merges path-name components into a single, full path name
perror	Prints error message
putenv	Adds or modifies value of environment variable

rand	Gets a pseudorandom number
_rotl	Shifts an **int** item to the left
_rotr	Shifts an **int** item to the right
_searchenv	Searches for a given file on specified paths
setjmp	Saves a stack environment
_splitpath	Splits a path name into component pieces
srand	Initializes pseudorandom series
swab	Swaps bytes of data

The "miscellaneous" category covers a number of commonly used routines that do not fit easily into any of the other categories. All routines except **assert**, **longjmp**, and **setjmp** are declared in **stdlib.h**. The **assert** routine is a macro and is defined in **assert.h**. The **setjmp.h** and **longjmp.h** functions are declared in **setjmp.h**.

The **abs** and **labs** functions return the absolute value of an **int** and a **long** value, respectively. These two functions are defined in both the **math.h** and **stdlib.h** include files.

The **div** and **ldiv** functions perform division of integers and long integers, respectively. They are both declared in **stdlib.h**.

The **assert** macro is typically used to test for program logic errors; it prints a message when a given "assertion" fails to hold true. Defining the identifier **NDEBUG** to any value causes occurrences of **assert** to be removed from the source file, thus allowing you to turn off assertion checking without modifying the source file.

The **getenv** and **putenv** routines provide access to the environment table. The global variable **environ** also points to the environment table, but it is recommended that you use the **getenv** and **putenv** routines to access and modify environment settings rather than accessing the environment table directly.

The **perror** routine prints the system error message, along with an optional user-supplied message, for the last system-level call that produced an error. The **perror** routine is declared in the include files **stdlib.h** and **stdio.h**. The error number is obtained from the **errno** variable. The system message is taken from the **sys_errlist** array. The **errno** variable is only guaranteed to be set upon error for those routines that explicitly mention the **errno** variable in the "Return Value" section of the reference pages in Part 2 of this manual.

The **rand** and **srand** functions initialize and generate a pseudorandom sequence of integers.

All four of the bit-shifting routines (**_lrotl**, **_lrotr**, **_rotl**, and **_rotr**) are declared in **stdlib.h**. They are used to shift bits of an interger or long integer to the left or right.

The **setjmp** and **longjmp** functions save and restore a stack environment. These routines let you execute a nonlocal goto.

The **swab** routine (also declared in **stdlib.h**) swaps bytes of binary data. It is typically used to prepare data for transfer to a machine that uses a different byte order. The **_makepath** routine combines the elements of a path name (drive, directory, file name, and extension) into a single "path-name" file. The **_splitpath** routine breaks up a "path-name" file into its component parts.

The **_searchenv** routine searches for a given file by examining a specified environment variable, such as **PATH**.

CHAPTER 5

INCLUDE FILES

5.1 Introduction

The include files provided with the run-time library contain macro and constant definitions, type definitions, and function declarations. Some routines require definitions and declarations from include files to work properly; for other routines, the inclusion of a file is optional. The description of each include file in this chapter explains the contents of each include file and lists the routines that use it.

A number of routines are declared in more than one include file. For example, the buffer-manipulation functions **memccpy**, **memchr**, **memcmp**, **memcpy**, **memicmp**, **memset**, and **movedata** are declared in both **memory.h** and **string.h**. These multiple declarations ensure agreement with the names of XENIX and UNIX include files, as well as with the names of include files under the proposed ANSI standard for C. Name agreement also preserves compatibility with programs written in earlier versions of C and further increases the portability of the programs you write in Microsoft C.

The include files were named and organized to meet the following objectives:

- To maintain compatibility with the names of include files on XENIX and UNIX systems, and with the ANSI standard for C
- To reflect the logical categories of run-time routines (for example, placing declarations for all memory-allocation functions in one file, **malloc.h**)
- To require inclusion of no more than the minimum number of files to use a given routine

Occasionally these goals conflict. For example, the **ftime** function uses the structure type **timeb**. The **timeb** structure type is defined in the include file **sys\timeb.h** on XENIX systems; to maintain compatibility, the same include file is used on MS-DOS. To minimize the number of required include files when using **ftime**, the **ftime** function is declared in **sys\timeb.h**, even though most of the other time functions are declared in **time.h**.

5.2 assert.h

The include file **assert.h** defines the **assert** macro. The **assert.h** file must be included when **assert** is used.

The definition of **assert** is enclosed in an **#ifndef** preprocessor block. If the identifier **NDEBUG** has not been defined (through a **#define** directive or on the compiler command line), the **assert** macro is defined to test a given expression (the "assertion"). If the assertion is false, a message is printed and the program is terminated.

If **NDEBUG** is defined, however, **assert** is defined as empty text. This disables all program assertions by removing all occurrences of **assert** from the source file. Therefore, you can suppress program assertions by defining **NDEBUG**.

5.3 bios.h

The **bios.h** include file contains functions declarations and structure definitions for the BIOS service routines, listed below:

_bios_disk
_bios_equiplist
_bios_keybrd
_bios_memsize
_bios_printer
_bios_serialcom
_bios_timeofday
int86
int86x

5.4 conio.h

The **conio.h** include file contains function declarations for all of the console and port I/O routines, as listed below:

cgets
cprintf
cputs
cscanf
getch
getche
inp
inpw
kbhit
outp
outpw
putch
ungetch

5.5 ctype.h

The **ctype.h** include file defines macros and constants and declares a global array used in character classification. The macros defined in **ctype.h** are listed below:

isalnum
isalpha
isascii
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
toascii
tolower
toupper
_tolower
_toupper

You must include **ctype.h** when using these macros or the macros will be undefined.

The **toupper** and **tolower** macros are defined as conditional operations. These macros evaluate their argument twice, and so produce unexpected results for arguments with side effects. To overcome this problem, you can remove the macro definitions of **toupper** and **tolower** and use the functions of the same names; see Section 4.3, "Character Classification and Conversion," for details. Declarations for the function versions of **tolower** and **toupper** are given in **stdlib.h**.

In addition to macro definitions, the **ctype.h** include file contains the following:

1. A set of manifest constants defined as bit masks. The bit masks correspond to specific classification tests. For example, the constants **_UPPER** and **_LOWER** are defined to test for an uppercase or lowercase letter, respectively.
2. A declaration of a global array, **_ctype**. The **_ctype** array is a table of character-classification codes based on ASCII character codes.

5.6 direct.h

The **direct.h** include file contains declarations for these functions:

chdir
getcwd
mkdir
rmdir

5.7 dos.h

The **dos.h** include file contains macro definitions, function declarations, and type definitions for the MS-DOS interface functions.

The **FP_SEG** and **FP_OFF** macros are defined to get or set the segment and offset portions of a far pointer. You must include **dos.h** when using these macros or they will be undefined.

The following functions are declared in **dos.h**:

bdos
_chain_intr
_disable
_dos_allocmem
_dos_close
_dos_creat
_dos_creatnew
_dos_findfirst
_dos_findnext
_dos_freemem
_dos_getdate
_dos_getdiskfree
_dos_getdrive
_dos_getfileattr
_dos_getftime
_dos_gettime
_dos_getvect
_dos_keep
_dos_open
_dos_read
_dos_setblock
_dos_setdate
_dos_setdrive
_dos_setfileattr
_dos_setftime
_dos_settime
_dos_setvect
_dos_write
dosexterr
_enable
_harderr
_hardresume
_hardretn
int86
int86x
intdos
intdosx
segread

The **dos.h** file also defines the **WORDREGS** and **BYTEREGS** structure types, used to define sets of word registers and byte registers, respectively. These structure types are combined in the **REGS** union type. The **REGS** union serves as a general-purpose register type, holding both register structures at one time. The **SREGS** structure type defines four members to hold the **ES**, **CS**, **SS**, and **DS** segment-register values.

The **DOSERROR** structure is defined to hold error values returned by the MS-DOS system call 0x59 (available under MS-DOS Versions 3.0 and later).

Note that **WORDREGS**, **BYTEREGS**, **REGS**, **SREGS**, and **DOSERROR** are tags, not **typedef** names. (See the *Microsoft C Language Reference* for a discussion of type definitions, tags, and **typedef** names.)

5.8 errno.h

The **errno.h** include file defines the values used by system-level calls to set the **errno** variable. The constants defined in **errno.h** are used by the **perror** function to index the corresponding error message in the global variable **sys_errlist**.

The constants defined in **errno.h** are listed with the corresponding error messages in Appendix A, "Error Messages."

5.9 fcntl.h

The include file **fcntl.h** defines flags used in the **open** and **sopen** calls to specify the type of operations for which the file is opened and to control whether the file is interpreted in text or binary mode. This file should always be included when **open** or **sopen** is used.

The function declarations for **open** and **sopen** are not in **fcntl.h**; instead, they are given in the include file **io.h**.

5.10 float.h

The include file **float.h** contains definitions of constants that specify the ranges of floating-point data types; for example, the maximum number of digits for objects of type **double** (**DBL_DIG** = 15), or the minimum exponent for objects of type **float** (**FLT_MIN_EXP** = –125).

The **float.h** file also contains function declarations for the math functions **_clear87**, **_control87**, **_fpreset**, and **_status87**, as well as definitions of constants used by these functions.

In addition, **float.h** defines floating-point-exception subcodes used with **SIGFPE** to trap floating-point errors (see Section 5.21, "signal.h").

5.11 graph.h

The **graph.h** include file declares all the routines in the graphics library, listed below:

_arc
_clearscreen
_displaycursor
_ellipse
_floodfill
_getbkcolor
_getcolor
_getcurrentposition
_getfillmask
_getimage
_getlinestyle
_getlogcoord
_getphyscoord
_getpixel
_gettextcolor
_gettextposition
_getvideoconfig
_imagesize
_lineto
_moveto
_outtext
_pie
_putimage
_rectangle
_remapallpalette
_remappalette
_selectpalette
_setactivepage
_setbkcolor
_setcliprgn
_setcolor
_setfillmask
_setlinestyle
_setlogorg
_setpixel
_settextcolor
_settextposition
_settextwindow
_setvideomode
_setviewport
_setvisualpage
_wrapon

It also defines several constants and structures used with the graphics routines. The manifest constants **_GBORDER** and **_GFILLINTERIOR** are used in the *control* parameter of the **_ellipse**, **_pie**, and **_rectangle** drawing routines. The **xycoord** structure stores position coordinates in pixels while the **rccoord** structure stores position coordinates in character rows and columns. The **videoconfig** structure stores information about the graphics hardware environment.

5.12 io.h

The include file **io.h** contains function declarations for most of the file-handling and low-level-I/O functions, as listed below:

access
chmod
chsize
close
creat
dup
dup2
eof
filelength
isatty
locking
lseek
mktemp
open
read
rename
setmode
sopen
tell
umask
unlink
write

The exceptions are **fstat** and **stat**, which are declared in **sys\stat.h**.

5.13 limits.h

The include file **limits.h** contains definitions of constants that specify the ranges of integer and character data types; for example, the maximum value for an object of type **char** (**CHAR_MAX** = 127).

5.14 malloc.h

The include file **malloc.h** contains function declarations for the memory-allocation functions listed below:

alloca
○**calloc**
_expand
_ffree
_fheapchk
_fheapset
_fheapwalk
_fmalloc
_fmsize
○**free**
_freect
halloc
_heapchk[1]
_heapset[1]
_heapwalk[1]
hfree
○**malloc**
_memavl
_memmax
_msize
_nfree
_nheapchk
_nheapset
_nheapwalk
_nmalloc
_nmsize
○**realloc**
sbrk
stackavail

---

[1] Implemented as a macro.

2 ○ strdup

The **malloc.h** file also contains the type definition for the structure **_heapinfo**, as well as several manifest constants used by the heap functions.

5.15 math.h

The include file **math.h** contains function declarations for all floating-point math routines, plus the **atof** routine, as listed below:

abs
acos
asin
atan
atan2
atof
bessel[1]
cabs
ceil
cos
cosh
exp
fabs
floor
fmod
frexp
hypot
labs
ldexp
log
log10
matherr
modf
pow
sin
sinh
sqrt
tan
tanh

The **math.h** include file also defines two structures, **exception** and **complex**. The **exception** structure is used with the **matherr** function, and the **complex** structure is used to declare the argument to the **cabs** function.

The HUGE_VAL value is returned on error from some math routines. For compatibility with XENIX, HUGE is defined as the equivalent of HUGE_VAL; both are defined in **math.h**. HUGE and HUGE_VAL may be implemented either as manifest constants or as global variables with **double** type and can be used interchangeably. The value of HUGE_VAL or HUGE must not be changed in a **#define** directive. Throughout Part 2, "Reference," references to HUGE_VAL are understood to mean either HUGE or HUGE_VAL.

The **math.h** file also defines manifest constants passed in the **exception** structure when a math routine generates an error (for example, DOMAIN, SING, EDOM, and ERANGE).

5.16 memory.h

The include file **memory.h** contains function declarations for the seven buffer-manipulation routines listed below:

---

[1] The **bessel** routine does not correspond to a single function but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

memccpy
memchr
memcmp
memcpy
memicmp
memset
movedata

5.17 process.h

The include file **process.h** declares all process-control functions (listed below) except for the **signal** function, which is declared in **signal.h**:

abort
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe
exit
_exit
getpid
spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe
system

The **process.h** include file also defines flags used in calls to **spawn** functions to control execution of the child process. Whenever you use one of the eight **spawn** functions, you must include **process.h** so the flags are defined.

5.18 search.h

The include file **search.h** declares the functions **bsearch**, **lsearch**, **lfind**, and **qsort**.

5.19 setjmp.h

The include file **setjmp.h** contains function declarations for the **setjmp** and **longjmp** functions. It also defines the machine-dependent buffer, **jmp_buf**, used by the **setjmp** and **longjmp** functions to save and restore the program state.

5.20 share.h

The include file **share.h** defines flags used in the **sopen** function to set the sharing mode of a file. This file should be included whenever **sopen** is used. The function declaration for **sopen** is given in the file **io.h**. Note that the **sopen** function should only be used under MS-DOS Versions 3.0 and later.

5.21 signal.h

The include file **signal.h** defines the values for the **SIGABRT**, **SIGINT**, **SIGFPE**, **SIGILL**, **SIGSEGV**, and **SIGTERM** signals.

C 4.0 Difference

> Microsoft C, Version 4.0, doesn't recognize the **SIGABRT**, **SIGILL**, and **SIGSEGV** signals.

The **signal** and **raise** functions are also declared in **signal.h**.

5.22 stdarg.h

The include file **stdarg.h** defines macros that allow you to access arguments in functions with variable-length argument lists, such as **vprintf**. These macros are defined to be machine independent, portable, and compatible with the developing ANSI standard for C. (Also see Section 5.31, **varargs.h**.)

5.23 stddef.h

The include file **stddef.h** contains definitions of the commonly used variables and types listed below:

Item	Description
NULL	The null pointer (also defined in **stdio.h**)
errno	A global variable containing an error message number (also defined in **errno.h**)
ptrdiff_t	Synonym for the type (**int**) of the difference between two pointers
size_t	Synonym for the type (**unsigned int**) of the value returned by **sizeof**

5.24 stdio.h

The include file **stdio.h** contains definitions of constants, macros, and types, along with function declarations for stream I/O functions. The stream I/O functions are listed below:

bsearch
calloc
clearerr
fclose
fcloseall
fdopen
feof[1]
ferror[1]
fflush
fgetc
fgetchar
fgetpos
fgets
fileno[1]
flushall
fopen
fprintf
fputc
fputchar
fputs
fread
freopen
fscanf
fseek
fsetpos
ftell
fwrite
getc[1]
getchar[1]
gets
getw
perror
printf
putc[1]
putchar[1]
puts
putw
qsort
remove
rename
rewind
rmtemp
scanf
setbuf
setvbuf
tempnam
tmpfile
tmpnam
ungetc
vfprintf
vprintf
vsprintf

The **stdio.h** file defines a number of constants; some of the more common ones are listed below:

Item	Description
BUFSIZ	Buffers used in stream I/O are of size **BUFSIZ** by default. This value is generally used to establish the size of system-allocated buffers. It is also required when you call **setbuf** to allocate your own buffers.
_NFILE	The **_NFILE** constant defines the number of open files allowed at one time. The files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** are always open, so you should include them when calculating the number of files your program opens.

---

[1] Implemented as a macro.

EOF	The **EOF** value is defined to be the value returned by an I/O routine when the end of the file (or in some cases, an error) is encountered.
NULL	The **NULL** value is the null-pointer value. It is defined as 0 in small- and medium-model programs and as 0L in large-model programs.

You can use the above constants in your programs, but you should not alter their values.

The **stdio.h** file also defines a number of flags used internally to control stream operations.

The **FILE** structure type is defined in **stdio.h**. Stream routines use a pointer to the **FILE** type to access a given stream. The system uses the information in the **FILE** structure to maintain the stream.

The **FILE** structures are stored as an array called **_iob**, with one entry per file. Therefore, each element of **_iob** is a **FILE** structure corresponding to a stream. When a stream is opened, it is assigned the address of an entry in the **_iob** array (a **FILE** pointer). Thereafter, the pointer is used for references to the stream.

5.25 stdlib.h

The **stdlib.h** include file contains function declarations for the following functions:

abort
abs
atexit
atof
atoi
atol
bsearch
calloc
div
ecvt
exit
_exit
fcvt
free
gcvt
getenv
itoa
labs
ldiv
_lrotl
_lrotr
ltoa
_makepath
malloc
max
min
onexit
perror
putenv
qsort
rand
realloc
_rotl
_rotr
_searchenv
_splitpath
srand
strtod
strtol
strtoul
swab
system
tolower
toupper
ultoa

The **tolower** and **toupper** routines are functions in the run-time library, but they are also implemented as macros in the include file **ctype.h**. The declarations for **tolower** and **toupper** are enclosed in an **#ifndef** block; they take effect only if the corresponding macro definitions in **ctype.h** have been suppressed by removing the definitions of **tolower** and **toupper**. For instructions on using these routines as macros or as functions, see Section 4.3, "Character Classification and Conversion."

The **stdlib.h** file also includes the definition of the type **onexit_t**, as well as declarations of the following global variables:

_doserrno
environ
errno
_fmode
_osmajor
_osminor
_osmode
_osversion
_psp
sys_errlist
sys_nerr

5.26 string.h

The **string.h** include file declares the string-manipulation functions, as listed below:

memccpy
memchr
memcmp
memcpy
memicmp
memmove
memset
movedata
strcat
strchr
strcmp
strcmpi
strcpy
strcspn
strdup
strerror
stricmp
strlen
strlwr
strncat
strncmp
strncpy
strnicmp
strnset
strpbrk
strrchr
strrev
strset
strspn
strstr
strtok
strupr

5.27 sys\locking.h

The **locking.h** include file (conventionally stored in a subdirectory named **sys**) contains definitions of flags used in calls to **locking**. Whenever you use the **locking** routine, you must include this file so that the locking flags are defined.

The function declaration for **locking** is given in the file **io.h**. Note that the **locking** function should be used only under MS-DOS Versions 3.0 and later.

5.28 sys\stat.h

The **stat.h** include file (conventionally stored in a subdirectory named **sys**) defines the structure type returned by the **fstat** and **stat** functions and defines flags used to maintain file-status information. It also contains function declarations for the **fstat** and **stat** functions. Whenever you use the **fstat** or **stat** functions, you must include this file so that the appropriate structure type (named **stat**) is defined.

5.29 sys\timeb.h

The include file **timeb.h** (conventionally stored in a subdirectory named **sys**) defines the **timeb** structure type and declares the **ftime** function, which uses the **timeb** structure type. Whenever you use the **ftime** function you must include **timeb.h** so that the structure type is defined.

5.30 sys\types.h

The include file **types.h** (conventionally stored in a subdirectory named **sys**) defines types used by system-level calls to return file-status and time information. You must include this file whenever the **sys\stat.h**, **sys\utime.h**, or **sys\timeb.h** file is included.

5.31 sys\utime.h

The include file **utime.h** (conventionally stored in a subdirectory named **sys**) defines the **utimbuf** structure type and declares the **utime** function, which uses the **utimbuf** type. Whenever you use the **utime** function you must include **utime.h** so that the structure type is defined.

5.32 time.h

The **time.h** include file declares the following time functions:

asctime
clock
ctime
difftime
gmtime
localtime
mktime
_strdate
_strtime
time
tzset

The **ftime** and **utime** functions are declared in **sys\timeb.h** and **sys\utime.h**, respectively.

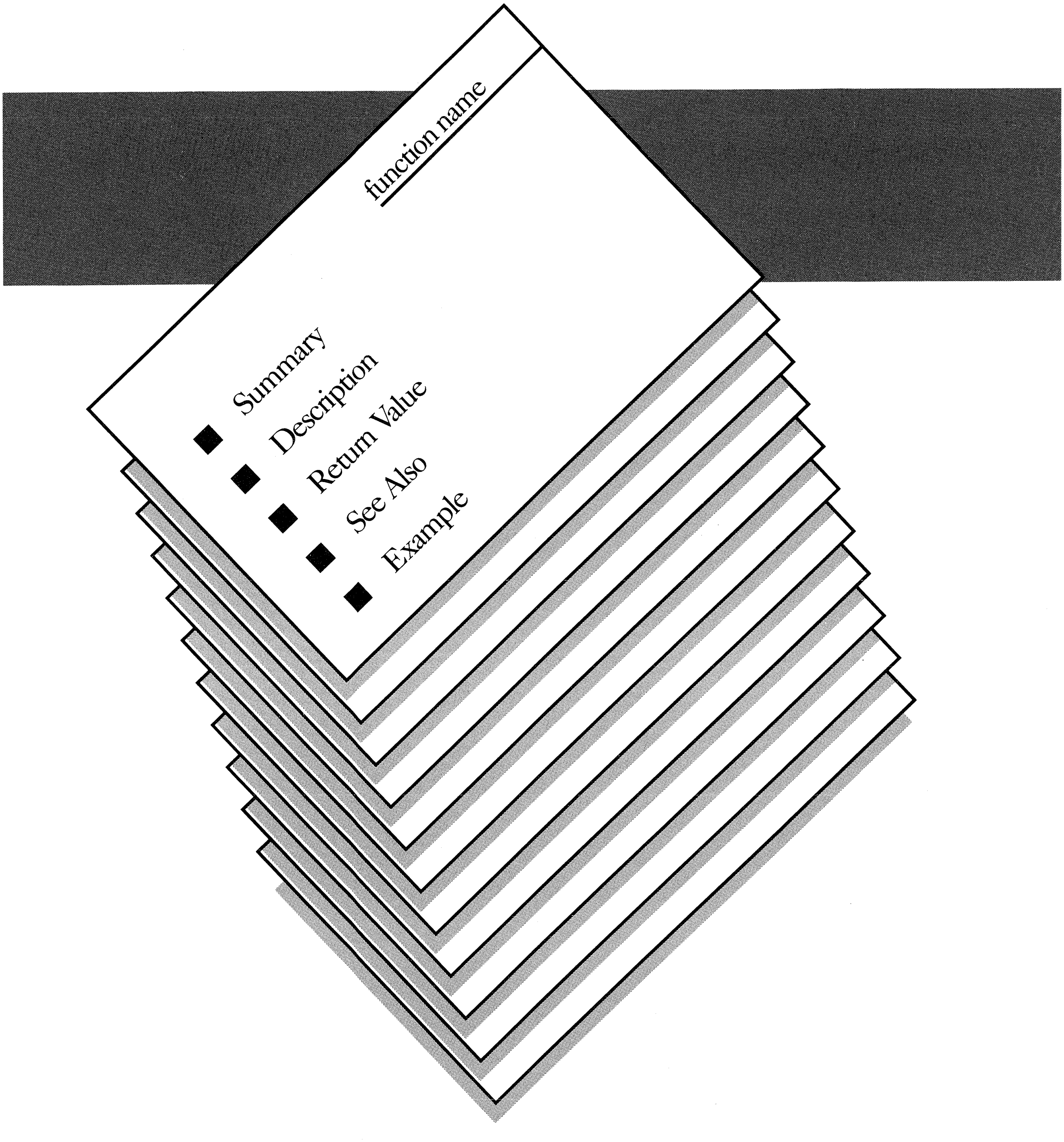
The **time.h** file also defines both the **tm** structure, used by the **asctime**, **gmtime**, and **localtime** functions, and the **time_t** type, used by the **difftime** function.

5.33 varargs.h

The include file **varargs.h** defines macros for accessing arguments in functions with variable-length argument lists, such as **vprintf**. These macros are defined to be machine independent, portable, and compatible with UNIX System V. (See also Section 5.22 on **stdarg.h**.)

PART 2

REFERENCE


function name
Summary
Description
Return Value
See Also
Example

PART 2

◇ REFERENCE

The second part of this manual is the reference section. Each routine in the run-time library is described here in alphabetical order. In some cases, similar or related routines are clustered in the same description, with differences noted where appropriate.

Descriptions follow the format illustrated on the opposite page. Below the Name of the routine, the Summary shows an exact syntax model for it and the Description outlines its actual effects. The Return Value is often useful to test for error conditions before using the results of a function call. See Also lists similar or related routines. The Example shows how the routine is used.

abort

Summary

#include <process.h>	Required only for function declarations
#include <stdlib.h>	Use either **process.h** or **stdlib.h**

void abort(void);

Description

The **abort** function prints the message

```
Abnormal program termination
```

to **stderr**, then calls **raise(SIGABRT)**. The action taken in response to the **SIGABRT** signal depends on what action has been defined for that signal in a prior call to the **signal** function. The default **SIGABRT** action is for the calling process to terminate with exit code 3, returning control to the parent process or operating system.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, **abort** prints the termination message and then terminates, without calling **raise(SIGABRT)**.

The **abort** function does not flush stream buffers or do **atexit/onexit** processing.

Return Value

By default, **abort** returns an exit code of 3 to the parent process or operating system.

See Also

exec functions, **exit**, **_exit**, **raise**, **signal**, **spawn** functions

Example

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
FILE *stream;
if ((stream = fopen(argv[argc-1],"r")) == NULL)
     {
     fprintf(stderr,
          "%s couldn't open file %s\n",argv[0],argv[argc-1]);
     abort();
     }
}

/* Note: the program name is stored in argv[0] only in
** DOS versions 3.0 and later; in versions prior to
** 3.0, argv[0] contains the string "C"
*/
```

Sample command line:

```
update employ.dat
```

Output:

```
C:\BIN\UPDATE.EXE couldn't open file employ.dat

Abnormal program termination
```

This program opens the file named on the command line for stream I/O. If this attempt fails, the program writes an error message to **stderr** and aborts.

Summary

# include <stdlib.h>	Required only for function declarations
int abs(*n*);	
int *n*;	Integer value

Description

The **abs** function returns the absolute value of its integer argument *n*.

Return Value

The **abs** function returns the absolute value of its argument. There is no error return.

See Also

cabs, fabs, labs

Example

```
#include <stdlib.h>

main()
 {
  int x = -4, y;

  y = abs(x);
  printf("%d\t%d\n",x,y);
 }
```

Output:

```
-4      4
```

This program computes and displays the absolute value of –4.

access

Summary

#include <io.h>	Required only for function declarations
int access(*path*, *mode*);	
char ****path*;**	File or directory path name
int *mode***;**	Permission setting

Description

With files, the **access** function determines whether or not the specified file exists and can be accessed in *mode*. The possible mode values and their meanings in the **access** call are as follows:

Value	Meaning
00	Check for existence only
02	Check for write permission
04	Check for read permission
06	Check for read and write permission

With directories, **access** determines only whether the specified directory exists; under MS-DOS, all directories have read and write access.

Return Value

The **access** function returns the value 0 if the file has the given mode. A return value of –1 indicates that the named file does not exist or is not accessible in the given mode, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Access denied: the file's permission setting does not allow the specified access.
ENOENT	File or path name not found.

See Also

chmod, **fstat**, **open**, **stat**

Example

```
#include <io.h>
#include <fcntl.h>
#include <stdio.h>

int fh;

main()
 {

 /* check for write permission:*/
 if ((access("data", 2)) == -1 )
   {
    perror("Data file not writable");
    exit(1);
   }
  else
   {
    fh = open("data",O_WRONLY);
    printf("Data file writable and opened for output");
   }
 }
```

This example uses **access** to check the file named `data` to see if writing is allowed.

acos

Summary

#include <math.h>

double acos(*x*);	
double *x*;	Value whose arccosine is to be calculated

Description

The **acos** function returns the arccosine of *x* in the range 0 to π. The value of *x* must be between –1 and 1.

Return Value

The **acos** function returns the arccosine result. If *x* is less than –1 or greater than 1, **acos** sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0. Error handling can be modified with the **matherr** routine.

See Also

asin, **atan**, **atan2**, **cos**, **matherr**, **sin**, **tan**

Example

```
#include <math.h>
#include <stdio.h>

extern int errno;

main()
 {
  float x, y;

  for (errno = EDOM;  errno == EDOM; y = acos(x))
   {
    printf("Cosine = ");
    scanf("%f", &x);
    errno = 0;
   }
  printf("Arccosine of %f = %f\n",x,y);
 }
```

This program prompts for input until it gets a value in the range –1 to 1. Input values outside this range produce an error message. When a correct value is entered, the program prints the arccosine of that value.

alloca

Summary

# include <malloc.h>	Required only for function declarations
void *alloca(*size***);**	
size_t *size***;**	Bytes to be allocated from stack

Description

The **alloca** routine allocates *size* bytes from the program's stack. The allocated space is automatically freed when the calling function is exited.

Return Value

The **alloca** routine returns a **char** pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **char**, use a type cast on the return value. The return value is **NULL** if the space cannot be allocated.

See Also

calloc, **malloc**, **realloc**

Warning

> The pointer value returned by **alloca** should never be passed as an argument to **free**, nor should **alloca** be used in an expression that is an argument to a function.

Example

```
#include <malloc.h>

main()
{
 int *intarray;
 intarray = (int *)alloca(10*sizeof(int));
}
```

This example calls **alloca** to allocate enough stack space for 10 integers.

Summary

#include <graph.h>

short far _arc (*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)	
short *x1*, *y1*;	Upper-left corner of bounding rectangle
short *x2*, *y2*;	Lower-right corner of bounding rectangle
short *x3*, *y3*;	Start vector
short *x4*, *y4*;	End vector

Description

The **_arc** function draws an elliptical arc. The center of the arc is the center of the bounding rectangle defined by the logical points (*x1*, *y1*) and (*x2*, *y2*). The arc starts at the point where it intersects the vector defined by (*x3*, *y3*) and ends where it intersects the vector defined by (*x4*, *y4*).

The arc is drawn using the current color, moving in a counterclockwise direction. Since an arc does not define a closed area, it is not filled.

Return Value

The **_arc** function returns a nonzero value if the arc is successfully drawn; otherwise, it returns 0.

See Also

_ellipse, **_lineto**, **_pie**, **_rectangle**, **_setcolor**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 _setvideomode(_MRES16COLOR);
 _arc( 80, 50, 240, 150, 0, 50, 240, 150 );
 while (!kbhit());       /* strike any key to clear screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws the arc shown in Figure R.1.

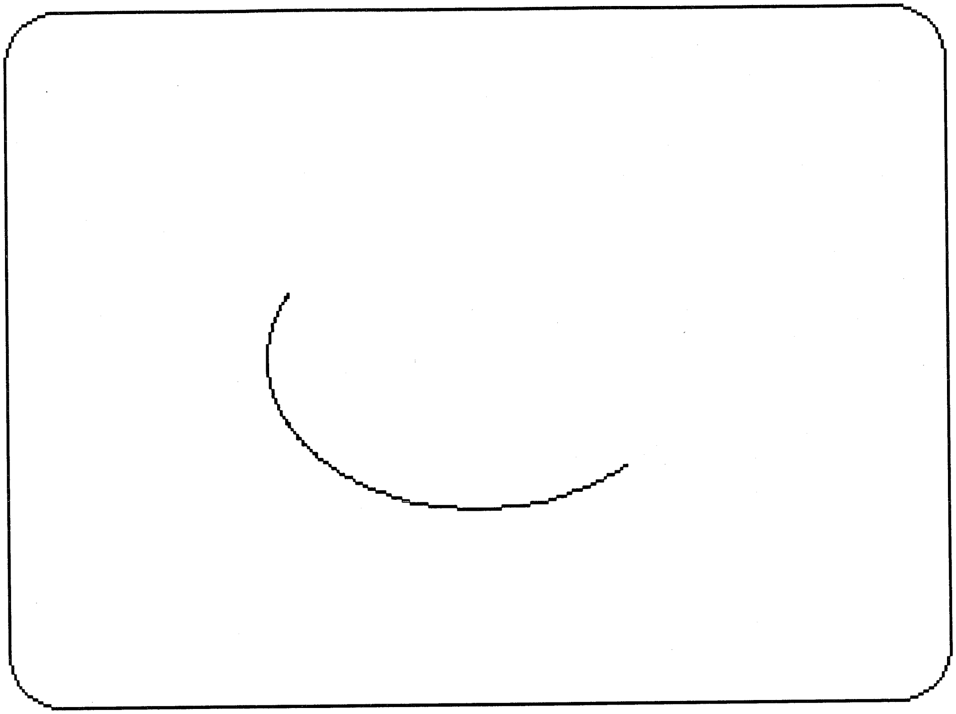


Figure R.1 Output of _arc Program

Summary

```
#include <time.h>

char *asctime(time);
struct tm {                 Time/date structure:
        int tm_sec;         Seconds after the minute (0–59)
        int tm_min;         Minutes after the hour (0–59)
        int tm_hour;        Hours since midnight (0–23)
        int tm_mday;        Day of the month (0–31)
        int tm_mon;         Months since January (0–11)
        int tm_year;        Years since 1900
        int tm_wday;        Days since Sunday (0–6)
        int tm_yday;        Days since January 1 (0–365)
        int tm_isdst;       Daylight-saving-time flag
        } tm *time;
```

Description

The **asctime** function converts a time stored as a structure to a character string. The *time* value is usually obtained from a call to **gmtime** or **localtime**, both of which return a pointer to a **tm** structure, defined in **time.h**. (See **gmtime** for a description of the **tm** structure fields.)

The string result produced by **asctime** contains exactly 26 characters and has the form of the following example:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The new-line character (**'\n'**) and the null character (**'\0'**) occupy the last two positions of the string.

Return Value

The **asctime** function returns a pointer to the character string result. There is no error return.

See Also

ctime, **ftime**, **gmtime**, **localtime**, **time**, **tzset**

Note

The **asctime** and **ctime** functions use a single statically allocated buffer to hold the return string. Each call to one of these routines destroys the result of the previous call.

Example

```
#include <time.h>
#include <stdio.h>

struct tm *newtime;
time_t aclock;

main()
 {
  time(&aclock);                   /* get time in seconds */

  /* Convert time to struct tm: */

  newtime = localtime(&aclock);
  printf("the current date and time are %s\n",

  /* Print local time as a string: */

  asctime(newtime));
 }
```

This program places the system time in the long integer `clock`, translates it into the structure `tm`, and then converts it to string form for output, using **asctime**.

Summary

#include <math.h>

double asin(*x*);	
double *x*;	Value whose arcsine is to be calculated

Description

The **asin** function calculates the arcsine of x in the range $-\pi/2$ to $\pi/2$. The value of x must be between –1 and 1.

Return Value

The **asin** function returns the arcsine result. If x is less than –1 or greater than 1, **asin** sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

Error handling can be modified by using the **matherr** routine.

See Also

acos, **atan**, **atan2**, **cos**, **matherr**, **sin**, **tan**

Example

```
#include <math.h>
#include <stdio.h>

extern int errno;

main()
 {
  float x, y;
  for (errno = EDOM;  errno == EDOM; y = asin(x))
   {
    printf("Sine = ");
    scanf("%f", &x);
    errno = 0;
   }
  printf("Arcsine of %f = %f\n",x,y);
 }
```

This program prompts for input until the input is in the range –1 to 1. If the input is outside this range, the program displays an error message. When correct input is entered, the program prints the arcsine of the input value.

Summary

#include <assert.h>
#include <stdio.h>

void assert(*expression***);**

Description

The **assert** routine prints a diagnostic message and calls the **abort** routine if *expression* is false (0). The diagnostic message has the form

`Assertion failed:` *expression,* `file` *filename,* `line` *linenumber*

where *filename* is the name of the source file and *linenumber* is the line number of the assertion that failed in the source file. No action is taken if *expression* is true (nonzero).

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, **assert** doesn't display *expression* in the diagnostic message.

The **assert** routine is typically used to identify program logic errors. The given expression should be chosen so that it holds true only if the program is operating as intended. After a program has been debugged, the special "no debug" identifier **NDEBUG** can be used to remove **assert** calls from the program. If **NDEBUG** is defined (by any value) with a **/D** command-line option or with a **#define** directive, the C preprocessor removes all **assert** calls from the program source.

The **assert** routine is implemented as a macro.

Return Value

There is no return value.

See Also

abort, **raise**, **signal**

Example

```
#include <stdio.h>
#include <assert.h>

analyze_string (string, length)
char *string;
int length;
 {
  assert(string != NULL);      /* Cannot be NULL */
  assert(*string != '\0');     /* Cannot be empty */
  assert(length > 0);          /* Length must be positive */

  printf( "Passed assertions.\n" );
 }

main()
 {
  analyze_string( "abc", 3 );
  analyze_string( "",    0 );
 }
```

In this program, the `analyze_string` function uses the `assert` function to test several conditions related to `string` and `length`. If any of the conditions fails, the program prints a message indicating what caused the failure.

Summary

#include <math.h>

double atan(*x*);	Calculate arctangent of *x*
double atan2(*y*, *x*); **double *x*, *y*;**	Calculate arctangent of *y*/*x*

Description

The **atan** and **atan2** functions calculate the arctangent of *x* and *y*/*x*, respectively: **atan** returns a value in the range –π/2 to π/2; **atan2** returns a value in the range –π to π. The **atan2** function uses the signs of both arguments to determine the quadrant of the return value.

Return Value

Both **atan** and **atan2** return the arctangent result. If both arguments of **atan2** are 0, the function sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

Error handling can be modified by using the **matherr** routine.

See Also

acos, **asin**, **cos**, **matherr**, **sin**, **tan**

Example

```
#include <math.h>
#include <stdio.h>

main()
{
  printf("%.7f\n",atan(1.0));        /* π/4 */
  printf("%.7f\n",atan2(-1.0,1.0);   /* -π/4 */
}
```

This program calculates and displays the arctangent of 1 and –1.

atexit

Summary

# include <stdlib.h>	Required only for function declarations
int atexit(*func*);	
void (func*)(void);**	Function to be called

Description

The **atexit** function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to **atexit** create a register of functions that are executed "last in, first out." No more than 32 functions can be registered with **atexit**, and it returns the value **NULL** if the number of functions exceeds 32. The functions passed to **atexit** cannot take parameters.

Return Value

The **atexit** function returns 0 if successful, or a nonzero value if not (e.g., there are already 32 exit functions defined).

See Also

abort, **exit**, **_exit**, **onexit**

Example

This program establishes several functions to be executed at the conclusion of the program. It also demonstrates how these functions are executed last in, first out.

```
#include <stdlib.h>
#include <stdio.h>

main()
 {
  int fn1(void), fn2(void), fn3(void), fn4(void);
  atexit(fn1);
  atexit(fn2);
  atexit(fn3);
  atexit(fn4);
  printf("This is executed first.\n");
 }

int fn1()
 {
  printf("next.\n");
 }

int fn2()
 {
  printf("executed ");
 }

int fn3()
 {
  printf("is ");
 }

int fn4()
 {
  printf("This ");
 }
```

Output:

```
This is executed first.
This is executed next.
```

This program pushes four functions onto the stack of functions to be executed when **atexit** is called. When the program exits, these programs are executed on a last-in, first-out basis.

atof, atol

■ Summary

#include <math.h>	
#include <stdlib.h>	Use either **math.h** or **stdlib.h**
double atof(*string***);**	Converts *string* to **double**
const char ****string*;**	String to be converted
#include <stdlib.h>	Required only for function declarations
int atoi(*string***);**	Converts *string* to **int**
long atol(*string***);**	Converts *string* to **long**
const char ****string*;**	String to be converted

■ Description

These functions convert a character string to a double-precision floating-point value (**atof**), an integer value (**atoi**), or a long integer value (**atol**). The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character it cannot recognize as part of a number. This character may be the null character (**\0**) terminating the string.

The **atof** function expects *string* to have the following form:

⟦*whitespace*⟧⟦{+ | –}⟧⟦*digits*⟧⟦.*digits*⟧ ⟦{**d** | **D** | **e** | **E**}⟦*sign*⟧*digits*⟧

A *whitespace* consists of space and/or tab characters, which are ignored; *sign* is either + or –; and *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed decimal integer.

The **atoi** and **atol** functions do not recognize decimal points or exponents. The *string* argument for these functions has the form

⟦*whitespace*⟧⟦*sign*⟧*digits*

where *whitespace, sign,* and *digits* are exactly as described above for **atof**.

Return Value

Each function returns the **double**, **int**, or **long** value produced by interpreting the input characters as a number. The return value is 0 (0L for **atol**) if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

See Also

ecvt, **fcvt**, **gcvt**

Example

```
#include <math.h>
#include <stdio.h>

extern long atol();
main()
     {
     char *s; double x; int i; long l;

     s = "  -2309.12E-15";    /* test of atof */
     x = atof(s);
     printf("%e\t",x);

     s = "7.8912654773d210"   /* test of atof */
     x = atof(s);
     printf("%e\t",x);

     s = "  -9885";           /* test of atoi */
     i = atoi(s);
     printf("%d\t",i);

     s = "98854 dollars";     /* test of atol */
     l = atol(s);
     printf("%ld\n",l);
     }
```

Output:

```
-2.309120e-012 7.891265e+210  -9885     98854
```

This program shows how numbers stored as strings can be converted to numerical values using the **atof**, **atoi**, and **atol** functions. Note that the **extern** declaration is needed only if the include file **stdlib.h** is absent.

bdos

Summary

#include <dos.h>

int bdos(*dosfn, dosdx, dosal*);	
int *dosfn*;	Function number
unsigned int *dosdx*;	**DX** register value
unsigned int *dosal*;	**AL** register value

Description

The **bdos** function invokes the MS-DOS system call specified by *dosfn* after placing the values specified by *dosdx* and *dosal* in the **DX** and **AL** registers, respectively. The **bdos** function executes an INT 21H instruction to invoke the system call. When the system call returns, **bdos** returns the contents of the **AX** register.

The **bdos** function is intended to be used to invoke MS-DOS system calls that either take no arguments or only take arguments in the **DX** (**DH**, **DL**) and/or **AL** registers.

Return Value

The **bdos** function returns the value of the **AX** register after the system call has completed.

See Also

intdos, **intdosx**

Warning

> This call should *not* be used to invoke system calls that indicate errors by setting the carry flag. Since C programs do not have access to this flag, the status of the return value cannot be determined. The **intdos** function should be used in these cases.

Example

```
#include <dos.h>
{
char *buffer = "Enter file name:$";

     /* Call 9 prints a string terminated by "$" */
     /* AL is not needed, so 0 is used */

bdos(9, (unsigned)buffer,0);
}
```

This example calls MS-DOS function 9H (display string) to display a prompt. The prompt is the string that `buffer` points to. This example works correctly only in small- and medium-model programs.

bessel

Summary

#include <math.h>

double j0(*x*);

double j1(*x*);

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

double y0(*x*);

double y1(*x*);

double yn(*n*, *x*);

double *x*;	Floating-point value
int *n*;	Integer order

Description

The **j0**, **j1**, and **jn** routines return Bessel functions of the first kind—orders 0, 1, and *n*, respectively.

The **y0**, **y1**, and **yn** routines return Bessel functions of the second kind—orders 0, 1, and *n*, respectively. The argument *x* must be positive.

Return Value

These functions return the result of a Bessel function of *x*.

For **y0**, **y1**, or **yn**, if *x* is negative, the routine sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns –**HUGE_VAL**.

Error handling can be modified by using the **matherr** routine.

See Also

matherr

■ Example

```
#include <math.h>
#include <stdio.h>

main()
{
  double x, y, z;

  x = 2;
  y = j0(x);
  z = yn(3,x);
  printf("y = %f and z = %f", y, z);
}
```

This program sets `y` to the Bessel function of the first kind, order 0, and sets `z` to the Bessel function of the second kind, order *n*.

_bios_disk

Summary

```
#include <bios.h>

unsigned _bios_disk(service, diskinfo);
unsigned service;                   Disk function
struct diskinfo_t {                 Disk parameters:
        unsigned drive;             Drive number
        unsigned head;              Head number
        unsigned track;             Track number
        unsigned sector;            Start sector number
        unsigned nsectors;          Number of sectors to read,
                                    write, or compare
        void far *buffer;           Memory location to write
                                    to, read from, or compare
        } *diskinfo;
```

Description

The **_bios_disk** routine uses INT 0x13 to provide several disk-access functions. The *service* parameter selects the function desired, while the *diskinfo* structure provides the necessary parameters.

The *service* argument can be set to one of the following manifest constants:

Constant	Function
_DISK_RESET	Forces the disk controller to do a hard reset, preparing for floppy-disk I/O. This is useful after an error occurs in another operation, such as a read. If this service is specified, the *diskinfo* argument is ignored.
_DISK_STATUS	Obtains the status of the last disk operation. Status is returned in the high-order bits of the return value, as listed below:

Bits	Meaning
0x01**	Invalid request or a bad command
0x02**	Address mark not found
0x04**	Sector not found
0x05**	Reset failed
0x07**	Drive parameter activity failed
0x09**	DMA overrun
0x0A**	Bad sector flag detected
0x10**	Data read (ECC) error
0x11**	Corrected data read (ECC) error
0x20**	Controller failure
0x40**	Seek error
0x80**	Disk timed out or failed to respond
0xAA**	Drive not ready
0xBB**	Undefined error
0xCC**	Write fault on drive
0xE0**	Status error

If this service is specified, the *diskinfo* argument is ignored.

_DISK_READ

Reads one or more disk sectors into memory. This service uses all fields of the structure that *diskinfo* points to, as defined in the Summary. If no error occurs, the function returns 0 in the high-order byte and the number of sectors read in the low-order byte. If there is an error, the high-order byte will contain a set of status flags, as defined under **_DISK_STATUS** (above).

_DISK_WRITE	Writes data from memory to one or more disk sectors. This service uses all fields of the structure that *diskinfo* points to, as defined in the Summary. If no error occurs, the function returns 0 in the high-order byte and the number of sectors written in the low-order byte. If there is an error, the high-order byte will contain a set of status flags, as defined under **_DISK_STATUS** (above).
_DISK_VERIFY	Checks the disk to be sure the specified sectors exist and can be read. It also runs a CRC (cyclic redundancy check) test. This service uses all fields (except *buffer*) of the structure that *diskinfo* points to, as defined in the Summary. If no error occurs, the function returns 0 in the high-order byte and the number of sectors compared in the low-order byte. If there is an error, the high-order byte will contain a set of status flags, as defined under **_DISK_STATUS** (above).
_DISK_FORMAT	Formats the track specified by *diskinfo*. The *head* and *track* fields indicate the track to format. Only one track can be formatted in a single call. The *buffer* field points to a set of sector markers. The format of the markers depends on the type of disk drive; see the *Technical Reference Manual* for the IBM PC to determine the marker format. There is no return value.

■ Example

```
#include <conio.h>
#include <stdio.h>
#include <bios.h>
```

```
main()
 {
    unsigned status = 0;
    struct diskinfo_t disk_info;

    disk_info.drive    = 0;
    disk_info.head     = 10;   /* invalid head number */
    disk_info.track    = 1;
    disk_info.sector   = 2;
    disk_info.nsectors = 8;

    status = _bios_disk (_DISK_VERIFY, &disk_info);
    if (status == 0x400) {
       printf ("return value: %x\n", status);
       printf ("seek error\n");
    }
    printf ("hit return to try again with valid head number\n");
    getchar ();
    disk_info.head = 0;
    status = _bios_disk (_DISK_VERIFY, &disk_info);
    if (status != 0x400) {
       printf ("return value: %x\n", status);
       printf ("no seek error\n");
    }
 }
```

This program first attempts to verify a disk by using an invalid disk head number. After printing the return value error code, the program verifies the disk by using a valid disk head code.

_bios_equiplist

Summary

#include <bios.h>

unsigned _bios_equiplist(void);

Description

The **_bios_equiplist** routine uses INT 0x11 to determine what hardware and peripherals are currently installed on the machine.

Return Value

The function returns a set of bits indicating what is installed, as defined below:

Bits	Meaning
0	Any disk drive installed if true
2-3	System RAM in 4K blocks (16-64K)
4-5	Initial video mode
6-7	Disk drives installed (00 = drive 1, 01 = drive 2, etc.)
8	False (0) if and only if a DMA chip is installed
9-11	Number of RS232 serial ports
12	True (1) if and only if a game adapter is installed
13	True (1) if and only if a serial printer is installed
14-15	Number of printers installed

■ Example

```
#include <bios.h>
main()
  {
  unsigned equipment;
  unsigned diskettes;
  equipment = _bios_equiplist();
  if (equipment & 0001)   /* check for diskette bit */
     printf ("diskettes installed\n");
  else
     printf ("no diskettes installed\n");
  }
```

This program checks for the presence of diskettes.

_bios_keybrd

Summary

#include <bios.h>

unsigned _bios_keybrd(*service*);	
unsigned *service*;	Keyboard function desired

Description

The **_bios_keybrd** routine uses INT 0x16 to access the keyboard services. The *service* argument can be any of the following manifest constants:

Constant	Meaning
_KEYBRD_READ	Reads the next character read from the keyboard. If no character has been typed, the call will wait for one. If the low-order byte of the return value is nonzero, it contains the ASCII value of the character typed. The high-order byte contains the keyboard scan code for the character. See the *Technical Reference Manual* for the IBM PC for a list of keyboard scan codes.
_KEYBRD_READY	Checks to see if a keystroke is waiting to be read and, if so, reads it. The return value is 0 if no keystroke is waiting, otherwise the return value is the character waiting to be read, in the same format as the **_KEYBRD_READ** return. The **_KEYBRD_READY** service does not remove the waiting character from the input buffer, as does the **_KEYBRD_READ** service.

_KEYBRD_SHIFTSTATUS Returns the current shift-key (SHIFT) status in the low-order byte of the return value. Any combination of the following bits may be set:

Bit	Meaning if True
0	Right-most SHIFT key pressed
1	Left-most SHIFT key pressed
2	CTRL key pressed
3	ALT key pressed
4	SCROLL LOCK on
5	NUM LOCK on
6	CAPS LOCK on
7	In insert mode (INS)

■ Example

```
#include <bios.h>
main()
  {
  while ((_bios_keybrd(_KEYBRD_SHIFTSTATUS) & 0001) != 1)
     printf ("Use right SHIFT key to stop this message\n");
  printf ("right SHIFT key pressed\n");
  }
```

This program prints a message on the screen until the right SHIFT key is pressed.

_bios_memsize

Summary

#include <bios.h>

unsigned _bios_memsize(void);

Description

The **_biosmemsize** routine uses INT 0x12 to determine the total amount of memory available.

Return Value

The routine returns the total amount of installed memory in 1K blocks. The maximum return value is 640, representing 640K of main memory.

Example

```
#include <bios.h>
main()
  {
  unsigned memory;
  memory = _bios_memsize();
  printf ("The amount of memory is: %dK\n", memory);
  }
```

This program displays the amount of memory available.

_bios_printer

■ Summary

#include <bios.h>

unsigned _bios_printer(*service, printer, data***);**
unsigned *service*; Printer function desired
unsigned *printer*; Target printer port
unsigned *data*; Output data

■ Description

The **_bios_printer** routine uses INT 0x17 to perform printer output services. The *printer* argument specifies the affected printer, where 0 is LPT1, 1 is LPT2, and so on. The *service* argument can be any of the following manifest constants:

Constant	Meaning
_PRINTER_WRITE	Sends the low-order byte of *data* to the printer specified by the *printer* argument. The low-order byte of the return value indicates the printer status after the operation, as defined below:

Bit	Meaning if True
0	Printer timed out
1	Not used
2	Not used
3	I/O error
4	Printer selected
5	Out of paper
6	Acknowledge
7	Printer not busy

Constant	Meaning
_PRINTER_INIT	Initializes the selected printer. The *data* argument is ignored. The return value is the low-order status byte defined above.
_PRINTER_STATUS	Returns the printer status in the low-order status byte defined above.

Example

```
#include <bios.h>
#include <conio.h>
#include <stdio.h>
#define LPT1 0
main()
 {
    unsigned data = 36;
    unsigned status;

    printf ("place printer offline and press return\n");
    getchar();    /* wait until key pressed */
    status = _bios_printer (_PRINTER_STATUS, LPT1, data);
    printf ("status with printer offline: %x\n\n", status);
    printf ("press return to initialize printer\n");
    getchar();    /* wait until key pressed */
    status = _bios_printer (_PRINTER_INIT, LPT1, data);
    printf ("status after printer initialized: %x\n", status);
 }
```

This program checks the status of the printer attached to LPT1 when it is off line, then initializes the printer.

Summary

#include <bios.h>

unsigned _bios_serialcom(*service, serial_port, data***);**	
unsigned *service***;**	Communications service
unsigned *serial_port***;**	Serial port to use
unsigned *data***;**	Port configuration bits

Description

The **_bios_serialcom** routine uses INT 0x14 to provide serial communications services. The *serial_port* argument is set to 0 for **COM1**, to 1 for **COM2**, and so on. The *service* argument can be set to one of the following manifest constants:

Constant	Service
_COM_INIT	Sets the port to the parameters specified in the *data* argument
_COM_SEND	Transmits the *data* characters over the selected serial port
_COM_RECEIVE	Accepts an input character from the selected serial port
_COM_STATUS	Returns the current status of the selected serial port

The *data* argument is ignored if *service* is set to **_COM_RECEIVE** or **_COM_STATUS**. The *data* argument for **_COM_INIT** is created by ORing together one or more of the following constants:

Constant	Meaning
_COM_CHR7	7 data bits
_COM_CHR8	8 data bits
_COM_STOP1	1 stop bit
_COM_STOP2	2 stop bits
_COM_NOPARITY	No parity
_COM_EVENPARITY	Even parity

_COM_ODDPARITY	Odd parity
_COM_110	110 baud
_COM_150	150 baud
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1200 baud
_COM_2400	2400 baud
_COM_4800	4800 baud
_COM_9600	9600 baud

The default value of *data* is 1 stop bit, no parity, and 110 baud.

Note

> This function works only with IBM Personal Computers and true compatibles.

Return Value

The function returns a 16-bit integer whose high-order byte contains status bits. The meaning of the low-order byte varies, depending on the *service* value. The high-order bits are as follows:

Bit	Meaning if Set
15	Timed out
14	Transmission-shift register empty
13	Transmission-hold register empty
12	Break detected
11	Framing error
10	Parity error
9	Overrun error
8	Data ready

When *service* is **_COM_SEND**, bit 15 will be set if *data* could not be sent.

When *service* is **_COM_RECEIVE**, the byte read will be returned in the low-order bits if the call is successful. If an error occurs, at least one of the high-order bits will be set.

When *service* is **_COM_INIT** or **_COM_STATUS**, the low-order bits are defined as follows:

Bit	Meaning if Set
7	Receive-line signal detected
6	Ring indicator
5	Data-set ready
4	Clear to send
3	Change in receive-line signal detected
2	Trailing-edge ring indicator
1	Change in data-set ready status
0	Change in clear-to-send status

Example

```
#include <bios.h>
main()
  {
  unsigned com1_status;
  com1_status =  _bios_serialcom(_COM_STATUS,0,0);
  printf ("COM1 status: %x\n",com1_status);
  }
```

This program checks the status of serial port COM1.

_bios_timeofday

■ Summary

#include <bios.h>

unsigned _bios_timeofday(*service, timeval***);**

int *service***;**	Time function desired
long *timeval***;**	Clock count

■ Description

The **_bios_timeofday** routine uses INT 0x1A to get or set the current system clock count. The *service* argument can be either of the following manifest constants:

Constant	Meaning
_TIME_GETCLOCK	Copies the current value of the clock count to the location that *timeval* points to. If midnight has not passed since the last time the system clock was read or set, the function returns 0; otherwise, it returns 1.
_TIME_SETCLOCK	Sets the current value of the system clock to the value in the location that *timeval* points to. There is no return value.

■ Example

```
#include <bios.h>
main()
  {
  long i, begin_tick, end_tick;
  _bios_timeofday (_TIME_GETCLOCK, &begin_tick);
  printf ("beginning tick count: %lu\n", begin_tick);
  for (i = 1; i<= 500000; i++)
      ;
  _bios_timeofday (_TIME_GETCLOCK, &end_tick);
  printf ("ending tick count:    %lu\n", end_tick);
  printf ("elapsed ticks:        %lu\n", end_tick - begin_tick);
  }
```

This program gets the current system clock count before and after a "do-nothing" loop and displays the difference.

Summary

#include <stdlib.h>	For ANSI compatibility
#include <search.h>	Required only for function declarations

void *bsearch(*key, base, num, width,* (*compare*)());

**const void ****key*;	Object to search for
**const void ****base*;	Pointer to base of search data
size_t *num, width*;	Number and width of elements
int (**compare*)(*elem1, elem2*);	*compare* function
**const void ****elem1*, ******elem2*;	Array elements to compare

Description

The **bsearch** function performs a binary search of a sorted array of *num* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The **bsearch** function will call the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values:

Value	Meaning
Less than 0	*element1* less than *element2*
0	*element1* identical to *element2*
Greater than 0	*element1* greater than *element2*

Return Value

The **bsearch** function returns a pointer to the first occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**.

See Also

lfind, **lsearch**, **qsort**

Example

```
#include <search.h>
#include <string.h>
#include <stdio.h>

int qcompare();  /* declare a function for qsort's compare   */
int bcompare();  /* declare a function for bsearch's compare */

main (argc, argv)
int argc;
char **argv;
{
 char **result;
 char *key = "PATH";
 int i;
 /* Sort using Quicksort algorithm: */
 qsort((char *)argv,argc,sizeof(char *),qcompare);
 for (i=0;i<argc;++i)                /* Output sorted list */
  printf("%s\n", argv[i]);
  /* Find item that begins with "PATH" */
  /* using a binary search algorithm: */
 result = (char **)bsearch((char *)&key, (char *)argv, argc,
                           sizeof(char *), bcompare);
 if (result)
  printf("%s found\n", *result);
 else
  printf("PATH not found!\n");
 }

int qcompare (arg1, arg2)
char **arg1, **arg2;
{                       /* Compare all of both strings: */
 return(strcmp(*arg1,*arg2));
}

int bcompare (arg1, arg2)
char **arg1, **arg2;
{                       /* Compare to length of key: */
 return(strncmp(*arg1,*arg2,strlen(*arg1)));
 }
```

This program reads the command-line arguments, sorting them with **qsort**, and then uses **bsearch** to find the parameter starting with PATH.

Summary

# include <math.h>

double cabs(*z*);
struct complex {
  double *x*; Real component
  double *y*; Imaginary component
  } *z*;

Description

The **cabs** function calculates the absolute value of a complex number, which must be a structure of type **complex**. A call to **cabs** is equivalent to the following:

sqrt(*z.x*z.x* + *z.y*z.y***)**

Return Value

On overflow, **cabs** calls **matherr**, returns **HUGE_VAL**, and sets **errno** to **ERANGE**.

See Also

abs, **fabs**, **labs**

Example

```
#include <math.h>
#include <stdio.h>
main()
{
  struct complex number;
  number.x = 3.0;
  number.y = 4.0;
  double d = cabs(number);
  printf("The absolute value of 'number' is %f\n", d);
}
```

Using **cabs**, this program assigns the absolute value of `number` to `d`.

calloc

Summary

# include <stdlib.h>	For ANSI compatibility
# include <malloc.h>	Required only for function declarations
void *calloc(*n*, *size***);**	
size_t *n***;**	Number of elements
size_t *size***;**	Length in bytes of each element

Description

The **calloc** function allocates storage space for an array of *n* elements, each of length *size* bytes. Each element is initialized to 0.

Return Value

The **calloc** function returns a pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value. The return value is **NULL** if there is insufficient memory available, or if *n* or *size* is 0.

C 4.0 Difference

In Version 4.0 of Microsoft C, **calloc** allocates a zero-length item (that is, a header only) in the heap if *size* is 0. The resulting pointer can be passed to the **realloc** function to adjust the size at any time.

See Also

free, **halloc**, **hfree**, **malloc**, **realloc**

Example

```
#include <stdio.h>
#include <malloc.h>

long *lalloc;

main()
 {
  lalloc = (long *)calloc(40,sizeof(long));

  if ( lalloc != NULL )
    printf( "Allocation OK\n" );
  else
    printf( "calloc failed\n" );
 }
```

This program uses **calloc** to allocate space for 40 long integers. It initializes each element to 0.

ceil

Summary

#include <math.h>

double ceil(*x*);	
double *x*;	Floating-point value

Description

The **ceil** function returns a **double** value representing the smallest integer that is greater than or equal to *x*.

Return Value

The **ceil** function returns the **double** result. There is no error return.

See Also

floor, **fmod**

Example

```
#include <stdio.h>
#include <math.h>

main()
 {
  double y;                                      /* y is equal to 2.0 */

  y = ceil(1.05);
  printf("The ceil( 1.05) is %f\n",y);

  y = ceil(-1.05);                               /* y is equal to -1.0 */
  printf("The ceil(-1.05) is %f\n",y);
 }
```

In this program, the smallest value representing an integer that is greater than or equal to the value passed to **ceil** is assigned to `y`.

Summary

#include <conio.h>	Required only for function declarations
char *cgets(*str*);	
**char ****str*;	Storage location for data

Description

The **cgets** function reads a string of characters directly from the console and stores the string and its length in the location pointed to by *str*. The *str* must be a pointer to a character array. The first element of the array, *str*[**0**], must contain the maximum length (in characters) of the string to be read. The array must have enough elements to hold the string, a terminating null character ('\0'), and two additional bytes.

The **cgets** function continues to read characters until a carriage-return–line-feed combination (CR-LF) is read, or the specified number of characters is read. The string is stored starting at *str*[**2**]. If a CR-LF combination is read, it is replaced with a null character ('\0') before being stored. The **cgets** function then stores the actual length of the string in the second array element, *str*[**1**].

Return Value

The **cgets** function returns a pointer to the start of the string, which is at *str*[**2**]. There is no error return.

See Also

getch, **getche**

Example

```
#include <conio.h>
#include <stdio.h>

char buffer[82];
char *result;

main()
 {
  buffer[0] = 80;            /* Maximum number of characters */

  printf("Input line of text, followed by carrage return:\n ");

  result = cgets(buffer);  /* Input a line of text */

  printf("\nLine length = %d\nText = %s\n", buffer[1], result);
 }

/* "buffer[1]" contains the length;
**  "result" points to the start of the string
*/
```

This program creates a buffer and initializes the first byte to the size of the buffer – 2. Next, the program accepts an input string using **cgets** and displays the size and text of that string.

Summary

#include <dos.h>

void _chain_intr(void (*target*)());
interrupt far ****target*;** Target interrupt routine

Description

The **_chain_intr** routine is used for chaining one interrupt handler to another interrupt handler. When the target handler begins executing, the stack and registers appear as though the target had been invoked directly when the interrupt occurred. Since the ultimate return address for the interrupt sequence is already on the stack, chaining subsequent handlers rather than calling them individually keeps the stack correct for the subsequent handler's return.

See Also

_dos_getvect, **_dos_keep**, **_dos_setvect**

chdir

Summary

#include <direct.h>	Required only for function declarations
int chdir(*path***);**	
char ****path*;**	Path name of new working directory

Description

The **chdir** function changes the current working directory to the directory specified by *path.* The *path* argument must refer to an existing directory.

This function can change the current working directory on any drive; it cannot change the default drive. For example, if **A:\** is the default drive and **BIN** is the current working directory, the following call changes the current working directory for drive C:

```
chdir(c:\temp);
```

In this case, you must first call the **system** function to change the current default drive to C before you can change the current working directory to that drive.

Return Value

The **chdir** function returns a value of 0 if the working directory is successfully changed. A return value of –1 indicates an error; in this case **errno** is set to **ENOENT**, indicating that the specified path name could not be found.

See Also

mkdir, **rmdir**, **system**

■ Example

```
#include <direct.h>
#include <stdio.h>

main(argc, argv)
int  argc;
char *argv[];

 {
  int rtnval;

  if (rtnval = chdir(argv[1]))
    printf("Problem changing to directory %s",argv[1]);
  else
    printf("Change to directory %s was successful",argv[1]);
 }
```

This program uses **chdir** to emulate the MS-DOS **cd** command.

chmod

Summary

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                   Required only for function declarations

int chmod(path, pmode);
char *path;                       Path name of existing file
int pmode;                        Permission setting for file
```

Description

The **chmod** function changes the permission setting of the file specified by *path*. The permission setting controls read and write access to the file. The constant expression *pmode* contains one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in **sys\stat.h**. Any other values for *pmode* are ignored. When both constants are given, they are joined with the bitwise-OR operator (¦). The meaning of the *pmode* argument is as follows:

Value	Meaning
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD ¦ S_IWRITE	Reading and writing permitted

If write permission is not given, the file is made read only. Under MS-DOS, all files are readable; it is not possible to give write-only permission. Thus the modes **S_IWRITE** and **S_IREAD ¦ S_IWRITE** are equivalent.

Return Value

The **chmod** function returns the value 0 if the permission setting is successfully changed. A return value of –1 indicates an error; in this case, **errno** is set to **ENOENT**, indicating that the specified file could not be found.

See Also

access, **creat**, **fstat**, **open**, **stat**

Example

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

int result;
int savestderr;

main()
 {

  /* make file read only:*/
  result = chmod("data", S_IREAD);
  if (result == -1)
    perror("File not found");
  else
    printf("Mode changed successfully");
 }
```

This program uses **chmod** to change the mode of the file `data` to read only. It then displays a message indicating whether the mode was changed successfully.

chsize

Summary

# include <io.h>	Required only for function declarations
int chsize(*handle*, *size*);	
int *handle*;	Handle referring to open file
long *size*;	New length of file in bytes

Description

The **chsize** function extends or truncates the file associated with *handle* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters (**'\0'**) are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file are lost.

Return Value

The **chsize** function returns the value 0 if the file size is successfully changed. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Specified file is locked against access (MS-DOS Versions 3.0 and later only).
EBADF	Specified file is read only, or an invalid file handle.
ENOSPC	No space left on device.

See Also

close, **creat**, **open**

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>

#define MAXSIZE 32768L

int fh, result;
char buffer[BUFSIZ] = "Initialize the buffer to some value\n";

main()
{
 int i;
 unsigned int nbytes = BUFSIZ;
 /* Output data to the file: */
 fh = open("data",O_RDWR|O_CREAT, S_IREAD|S_IWRITE);

 for (i = 0; i < 50; i++)
  result = write(fh, buffer, nbytes);
 result = -1;
 if (lseek(fh,0L,SEEK_END) > MAXSIZE)    /* Make sure the file */
                                         /* is longer than 32k */
  result = chsize(fh,MAXSIZE);    /* before changing the size. */
 if (result == 0)
  printf("Size successfully changed");
 else
  printf("Problem in changing the size");
}
```

This program opens the file named `data` and writes data to it. Then it uses **chsize** to extend the size of `data`. Finally, it displays a message indicating whether the file size was successfully extended.

_clear87

Summary

#include <float.h>

unsigned int _clear87(void); Get and clear floating-point status word

Description

The **_clear87** function gets and clears the floating-point status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

Return Value

The bits in the value returned indicate the floating-point status. See the **float.h** include file for a complete definition of the bits returned by **_clear87**.

Note

Many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from **_clear87** and **_status87** become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

See Also

_control87, **_status87**

Example

```
#include <stdio.h>
#include <float.h>

double a = 1e-40,b;
float x,y;

main ( )
  {
  printf("status  = %.4x - clear\n",_clear87( ));

  /* store into y is inexact and underflows: */
  y = a;
  printf("status  = %.4x - inexact, underflow\n",_clear87( ));

  /* y is denormal: */
  b = y;
  printf("status  = %.4x - denormal\n",_clear87( ));
  }
```

This program creates various floating-point problems, then uses **_clear87** to report on these problems.

clearerr

Summary

#include <stdio.h>

void clearerr(*stream*);	
FILE **stream*;	Pointer to **FILE** structure

Description

The **clearerr** function resets the error indicator and end-of-file indicator for *stream* to 0. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr** or **rewind** is called.

See Also

eof, **feof**, **ferror**, **perror**

Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

main()
 {
  stream = fopen("data", "w");   /* Note that with "w"      */
  if ((c = getc(stream)) == EOF) /* there will be an error. */
   {
    if (ferror(stream))
     {
      fprintf(stderr,"Read error\n");
      clearerr(stream);
     }
   }
 }
```

This program sends data to a stream and checks to see whether an error has occurred. If so, the program uses **clearerr** to clear the error.

Summary

#include <graph.h>

void far _clearscreen(*area*);
short *area*; Target area

Description

The **_clearscreen** function erases the target area, filling it with the current background color. The *area* parameter can be one of the following manifest constants (defined in **graph.h**):

Constant	Action
_GCLEARSCREEN	Clears and fills the entire screen
_GVIEWPORT	Clears and fills only within the current view port
_GWINDOW	Clears and fills only within the current text window

Return Value

There is no return value.

See Also

_getbkcolor, **_setbkcolor**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int xvar, yvar, loop = 0;
 _setvideomode( _MRES16COLOR );
 /* Make 16 rectangles */
 for ( loop = 0; loop < 32; loop += 2 ) {
  _setcolor( loop % 16 );
  _rectangle(_GFILLINTERIOR, loop*10, 95, (loop+1)*10, 105);
 }
 while ( !kbhit() ) { /* Repeat until a character is typed */
  _remappalette( loop++ % 4, rand(1) % 16 );
 }
 _clearscreen( _GCLEARSCREEN );
 _setvideomode ( _DEFAULTMODE);
}
```

This program draws 16 separate rectangles, each of a different color. When it receives a keystroke, it calls **_clearscreen** and clears the screen.

Summary

#include <time.h>

clock_t clock(void);

Description

The **clock** function tells how much processor time has been used by the calling process. The time in seconds is approximated by dividing the **clock** return value by the value of the **CLK_TCK** macro.

Return Value

The **clock** function returns the product of the time in seconds and the value of the **CLK_TCK** macro. If the processor time is not available, the function returns the value -1, cast as **clock_t**.

See Also

difftime, **time**

Example

```
#include <stdio.h>
#include <time.h>

main()
{
 int goal, tm = 0;
 clock_t clock(void);
 printf("How many seconds do you want the program to run?: ");
 scanf("%d",&goal);
 do {
  if ((tm=clock()) != (clock_t)-1)
   printf("Processor time equals %d seconds\n", tm/CLK_TCK);
  else {
   printf("Processor time not available\n");
   exit(-1);
  }
 }
 while ((tm/CLK_TCK) < goal);
}
```

This example prompts for how long the program is to run and then continuously displays the elapsed time for that period.

close

Summary

#include <io.h>	Required only for function declarations
int close(*handle*);	
int *handle*;	Handle referring to open file

Description

The **close** function closes the file associated with *handle.*

Return Value

The **close** function returns 0 if the file was successfully closed. A return value of –1 indicates an error, and **errno** is set to **EBADF**, indicating an invalid file-handle argument.

See Also

chsize, **creat**, **dup**, **dup2**, **open**, **unlink**

Example

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>

main()
 {
  int result, fh;

  fh = open("data",O_RDONLY); /* Open the file */
  result = close(fh);         /* Now close it  */
  /* Report on results: */
  if (result)
    printf("Invalid file handle argument\n");
  else
    printf("File successfully closed\n");
 }
```

This program uses **open** to open a file named `data`, then uses **close** to close it.

Summary

#include <float.h>

unsigned int _control87(*new*, *mask***);**	Get floating-point control word
unsigned int *new***;**	New control-word bit values
unsigned int *mask***;**	Mask for new control-word bits to set

Description

The **_control87** function gets and sets the floating-point control word. The floating-point control word allows the program to change the precision, rounding, and infinity modes in the floating-point-math package. Floating-point exceptions can also be masked or unmasked using the **_control87** function.

If the value for *mask* is equal to 0, then **_control87** gets the floating-point control word. If *mask* is nonzero, then a new value for the control word is set in the following manner: for any bit that is on (equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. To put it another way,

```
fpcntrl = ((fpcntrl & ~mask) | (new & mask))
```

where `fpcntrl` is the floating-point control word.

Return Value

The bits in the value returned indicate the floating-point control state. See the **float.h** include file for a complete definition of the bits returned by **_control87**.

See Also

_clear87, **_status87**

Example

```
#include <stdio.h>
#include <float.h>

double a = .1;

main()
  {

   /* get control word: */
   printf("control = %.4x\n", _control87(0,0));
   printf("a*a = .01 = %.15e\n",a*a);

   /* set precision to 24 bits: */
   _control87(PC_24,MCW_PC);

   printf("a*a = .01 (rounded to 24 bits) = %.15e\n",a*a);

   /* restore to initial default: */
   _control87(CW_DEFAULT,0xffff);

   printf("a*a = .01 = %.15e\n",a*a);
  }
```

This program uses **_control87** to output the control word, set the precision to 24 bits, and reset the status to the default.

Summary

#include <math.h>

double cos(*x*);	Calculates cosine of *x*
double cosh(*x*);	Calculates hyperbolic cosine of *x*
double *x*;	Radians

Description

The **cos** and **cosh** functions return the cosine and hyperbolic cosine, respectively, of *x*.

Return Value

If *x* is large, a partial loss of significance in the result may occur in a **cos** call, in which case the function generates a **PLOSS** error. If *x* is so large that significance is completely lost, **cos** prints a **TLOSS** message to **stderr** and returns 0. In both cases, **errno** is set to **ERANGE**.

If the result is too large in a **cosh** call, the function returns **HUGE_VAL** and sets **errno** to **ERANGE**.

See Also

acos, **asin**, **atan**, **atan2**, **matherr**, **sin**, **sinh**, **tan**, **tanh**

Example

```
#define PI 3.14159265359
#include<math.h>
#include<stdio.h>

main()
{
 double x = cos( PI);          /* x = -1 */
 double y = cosh(PI);          /* y = 11.591953 */
 printf("The cos(PI) = %f and the cosh(PI) = %f\n",x,y);
}
```

This program displays the cosine and hyperbolic cosine of π.

cprintf

Summary

#include <conio.h> Required only for function declarations

int cprintf(*format*⟦**,** *argument*⟧**...);**
**char ****format*; Format control string

Description

The **cprintf** function formats and prints a series of characters and values directly to the console, using the **putch** function to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format.* The format has the same form and function as the *format* argument for the **printf** function; see the **printf** reference page for a description of the format and arguments.

Return Value

The **cprintf** function returns the number of characters printed.

See Also

fprintf, **printf**, **sprintf**, **vprintf**

Note

> Unlike the **fprintf**, **printf**, and **sprintf** functions, **cprintf** does not translate line-feed (LF) characters into carriage-return–line-feed (CR-LF) combinations on output.

Example

```
#include <conio.h>

int i = -16, j = 29;
unsigned int k = 511;

main()
 {
  cprintf("i=%d, j=%#x, k=%u\n",i,j,k);
  /* Output:  i=-16, j=0x1d, k=511 */

 }
```

This program prints the values of the variables `i`, `j`, and `k` to the console. (The **cprintf** function is similar to the **printf** function except that it sends output to the console.)

cputs

Summary

# include <conio.h>	Required only for function declarations
int cputs(*string*);	
char ****string*;**	Output string

Description

The **cputs** function writes the null-terminated string pointed to by *str* directly to the console. Note that a carriage-return–line-feed (CR-LF) combination is not automatically appended to the string after writing.

Return Value

If successful, **cputs** returns a 0. If the function fails, it returns a nonzero value.

C 4.0 Difference

In Version 4.0 of Microsoft C, **cputs** has no return value.

See Also

putch

Example

```
#include <conio.h>

  char *buffer = "Insert data disk in drive a: \r\n";

main()
 {
  cputs(buffer);
 }
```

This program displays on the console the prompt that `buffer` points to.

Summary

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                    Required only for function declarations

int creat(path, pmode);
char *path;                        Path name of new file
int pmode;                         Permission setting
```

Description

The **creat** function either creates a new file or opens and truncates an existing file. If the file specified by *path* does not exist, a new file is created with the given permission setting and is opened for writing. If the file already exists and its permission setting allows writing, **creat** truncates the file to length 0, destroying the previous contents, and opens it for writing.

The permission setting, *pmode,* applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. The integer expression *pmode* contains one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in **sys\stat.h**. When both constants are given, they are joined with the bitwise-OR operator (¦). The meaning of the *pmode* argument is as follows:

Value	Meaning
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD ¦ S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read only. Under MS-DOS it is not possible to give write-only permission. Therefore, the modes **S_IWRITE** and **S_IREAD ¦ S_IWRITE** are equivalent. Under MS-DOS Versions 3.0 and later, files opened using **creat** are always opened in compatibility mode (see **sopen**).

The **creat** function applies the current file-permission mask to *pmode* before setting the permissions (see **umask**).

Return Value

If successful, **creat** returns a handle for the created file. Otherwise, it returns –1 and sets **errno** to one of the following constants:

Value	Meaning
EACCES	Path name specifies an existing read-only file or specifies a directory instead of a file
EMFILE	No more handles available (too many open files)
ENOENT	Path name not found

See Also

chmod, **chsize**, **close**, **dup**, **dup2**, **open**, **sopen**, **umask**

Note

The **creat** routine is provided primarily for compatibility with previous libraries. A call to **open** with **O_CREAT** and **O_TRUNC** in the *oflag* argument is equivalent to **creat** and is preferable for new code.

Example

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
 int fh = creat("data",S_IREAD|S_IWRITE);
 if (fh == -1)
  perror("Couldn't create data file");
 else
  printf("Created data file.\n");
}
```

This program uses **creat** to create the file (or truncate the existing file) named `data` and open it for writing.

■ Summary

#include <conio.h>	Required only for function declarations
int cscanf(*format*⟦**,** *argument*⟧...**);**	
char ****format*;**	Format-control string

■ Description

The **cscanf** function reads data directly from the console into the locations given by the *arguments* (if any), using the **getche** function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function; see the **scanf** reference page for a description of *format*.

Note

While **scanf** normally echoes the input character, it will not do so if the last call was to **ungetch**.

■ Return Value

The **cscanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

■ See Also

fscanf, **scanf**, **sscanf**

Example

```
#include <conio.h>

int result;
char buffer[20];

main()
 {
  cprintf("Please enter file name: ");

  /* Read in user response; return # of matches: */
  result = cscanf("%19s",buffer);

  printf("\nNumber of correctly matched input "
         "items = %d\n", result );
 }
```

This program prompts for a file name and uses **cscanf** to read in the corresponding file. Then **cscanf** returns the number of items matched, and the program displays that number.

Summary

#include <time.h>	Required only for function declarations
char *ctime(*time*);	
const time_t ***time*;**	Pointer to stored time

Description

The **ctime** function converts a time stored as a **time_t** value to a character string. The *time* value is usually obtained from a call to **time**, which returns the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

The string result produced by **ctime** contains exactly 26 characters and has the form of the following example:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The new-line character (**'\n'**) and the null character (**'\0'**) occupy the last two positions of the string.

Return Value

The **ctime** function returns a pointer to the character string result. If *time* represents a date before 1980, **ctime** returns **NULL**.

C 4.0 Differences

MS-DOS does not understand dates prior to 1980. If *time* represents a date before January 1, 1980, the **ctime** routine in Version 4.0 of the Microsoft C Run-Time Library returns the character string representation of 00:00:00 January 1, 1980.

See Also

asctime, **ftime**, **gmtime**, **localtime**, **time**

Note

The **asctime** and **ctime** functions use a single statically allocated buffer for holding the return string. Each call to one of these routines destroys the result of the previous call.

Example

```
#include <time.h>
#include <stdio.h>

time_t ltime;

main()
 {
  time(&ltime);
  printf("the time is %s\n",ctime(&ltime));
 }
```

This program gets the current time in **time_t** form, then uses **ctime** to display the time in string form.

dieeetomsbin, dmsbintoieee

Summary

#include <math.h>

int dieeetomsbin(*src8*, *dst8***);**	IEEE double to MS binary double
int dmsbintoieee(*src8*, *dst8***);**	MS binary double to IEEE double
double ****src8*;**	Buffer containing value to convert
double ****dst8*;**	Buffer to store converted value

Description

The **dieeetomsbin** routine converts a double-precision number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft binary format. The **dmsbintoieee** routine converts a double-precision number in Microsoft binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random-access data files created with those versions of Microsoft BASIC that store floating-point numbers in Microsoft binary format, and vice versa.

The argument *src8* is a pointer to the **double** value to be converted. The result is stored at the location given by *dst8*.

Return Value

These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

See Also

fieeetomsbin, **fmsbintoieee**

Note

These routines do not handle IEEE NANs and infinities. IEEE denormals are treated as 0 in the conversions.

difftime

Summary

#include <time.h>	Required only for function declarations
double difftime(*time2*, *time1***);** **time_t** *time2*; **time_t** *time1*;	Type **time_t** defined in **time.h**

Description

The **difftime** function computes the difference *time2* – *time1*.

Return Value

The **difftime** function returns the elapsed time in seconds from *time1* to *time2* as a double-precision number.

See Also

time

Example

```
#include <time.h>

int mark[10000];

main()
{
 time_t  start, finish;
 register int i, loop, n, num, step;
 printf("This program will take about 3 minutes "
        "on an AT and 8 on a PC\n");
 printf("Working...\n");
 time(&start);
 for (loop = 0; loop < 1000; ++loop)
  for (num = 0,n = 3; n < 10000; n += 2)
   if (!mark[n]){     /* printf("%d\t",n); */
    step = 2*n;
    for (i = 3*n; i < 10000; i += step)
    mark[i] = -1;
    ++num;
   }
 time(&finish);
 /* Prints average of 1000 loops through "sieve": */
 printf("\nProgram takes %f seconds to find %d primes.\n",
        difftime(finish,start)/1000, num);
}
```

Output:

```
Program takes 0.482000 seconds to find 1228 primes.
```

This program calculates the amount of time needed to find the prime numbers between 3 and 10,000. To display the prime numbers, delete the outermost loop and the comment delimiters around the expression `printf("%d\t",n);`.

_disable

Summary

#include <dos.h>

void _disable(void); Disables interrupts

Description

The **_disable** routine disables interrupts by executing an 8086 CLI machine instruction.

See Also

_enable

Summary

#include <graph.h>

short far _displaycursor(*toggle*);
short *toggle*; Cursor state

Description

On entry into each graphic routine, the screen cursor is turned off. The **_displaycursor** function determines whether or not the cursor is to be turned back on when programs exit graphic routines. If *toggle* is set to **_GCURSORON**, the cursor will be restored on exit. If *toggle* is set to **_GCURSOROFF**, the cursor will be left off on exit.

Return Value

The function returns the previous value of *toggle*. There is no error return.

Example

```
#include <stdio.h>
#include <graph.h>

main()

{
  _setvideomode( _MRES4COLOR );
  _settextposition( 1, 1 );
  _displaycursor( _GCURSORON );
  _outtext( "Cursor on, hit <cr>" );
  for( ;!kbhit(); );
  getchar();
  _settextposition( 1, 1 );
  _displaycursor( _GCURSOROFF );
  _outtext( "Cursor off, hit <cr>" );
  for( ;!kbhit(); );
  getchar();
  _setvideomode( _DEFAULTMODE );
}
```

This program shows the effect of turning the cursor on and off in a graphics mode.

div

Summary

```
#include <stdlib.h>

struct div_t {
        int quot;                     Quotient
        int rem;                      Remainder
        } div(numer, denom);
int numer;                            Numerator
int denom;                            Denominator
```

Description

The **div** function divides *numer* by *denom*, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0 the program will terminate with an error message.

Return Value

The **div** function returns a structure of type **div_t**, comprising both the quotient and the remainder. The structure is defined in **stdlib.h**.

See Also

div

Example

```
#include <stdlib.h>
#include <math.h>

main(argc, argv)
int argc;
char **argv;
 {
  int x,y;
  div_t div_result;
  x = atoi(argv[1]);
  y = atoi(argv[2]);
  printf("x is %d, y is %d\n", x,y);
  div_result = div(x,y);
  printf("The quotient is %d, and the remainder is %d\n",
         div_result.quot, div_result.rem);
 }
```

The example above takes two integers as command-line arguments and displays the results of the integer division. This program accepts two arguments on the command line following the program name, then calls **div** to divide the first argument by the second. Finally, it prints the structure members *quot* and *rem.*

Assuming the executable file is named "tdiv," it might be typed:

```
tdiv 5 2
```

and it would output:

```
x is 5, y is 2
The quotient is 2, and the remainder is 1
```

_dos_allocmem

Summary

#include <dos.h>

unsigned _dos_allocmem(*size, segment***);**

unsigned *size***;**	Block size to allocate
unsigned ****segment*;**	Segment descriptor return buffer

Description

The **_dos_allocmem** function allocates a block of memory *size* paragraphs long. A paragraph is 16 bytes. Allocated blocks are always paragraph aligned. The segment descriptor for the initial segment of the new block is returned in the word that *segment* points to. If the request cannot be satisfied, the maximum possible size (in paragraphs) is returned in this word instead.

Return Value

If successful, **_dos_allocmem** returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to **ENOMEM**, indicating insufficient memory or invalid arena (memory area) headers.

See Also

alloca, **calloc**, **_dos_freemem**, **_dos_setblock**, **halloc**, **malloc**

Example

```
#include <dos.h>

unsigned segment;

main()
  {
  /* Allocate 20 paragraphs */
  if (_dos_allocmem (20, &segment) != 0)
     printf ("allocation failed\n");
  else
     printf ("allocation successful\n");
  if (_dos_freemem (segment) != 0)
     printf ("free memory failed\n");
  else
     printf ("free memory successful\n");
 }
```

This program allocates and then frees 20 paragraphs of memory space.

_dos_close

Summary

#include <dos.h>

unsigned _dos_close(*handle*);
int *handle*; Target file handle

Description

The **_dos_close** function uses system call 0x3E to close the file indicated by *handle*. The file's *handle* argument is returned by the call that created or last opened the file.

Return Value

The function returns 0 if successful. Otherwise, it returns the MS-DOS error code and sets **errno** to **EBADF**, indicating an invalid file handle.

See Also

creat, **_dos_creat**, **_dos_creatnew**, **_dos_open**, **_dos_read**, **_dos_write**, **dup**, **fclose**, **open**

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
```

```
main()
 {
  int fh;

  /* open file with _dos_open function */
  if (_dos_open("data1",O_RDONLY, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program uses the MS-DOS I/O operations to open and close a file.

_dos_creat, _dos_creatnew

Summary

```
#include <dos.h>

unsigned _dos_creat(path, attribute, handle);

unsigned _dos_creatnew(path, attribute, handle);

char *path;                 File path name
unsigned attribute;         File attributes
int *handle;                Handle return buffer
```

Description

The **_dos_creat** and **_dos_creatnew** routines create a new file named *path*, having the access attributes specified in the *attribute* word. The new file's handle is copied into the buffer that *handle* points to. The file is opened for both read and write access. If file sharing is installed, the file is opened in compatibility mode.

The **_dos_creat** routine uses system call 0x3C and the **_dos_creatnew** routine uses system call 0x5B. If the file already exists, **_dos_creat** will erase its contents and leave its attributes unchanged; however, the **_dos_creatnew** routine will fail if the file already exists.

Return Value

If successful, both routines return 0. Otherwise, they return the MS-DOS error code and set **errno** to one of the following values:

Constant	Meaning
ENOENT	Path or file not found
EMFILE	Too many open file handles
EACCES	Access denied because the directory is full or, for **_dos_creat** only, the file exists and cannot be overwritten
EEXIST	File already exists (**_dos_creatnew** only)

Example

```
#include <dos.h>

main()
{
 int fh1, fh2;
 if (_dos_creat("data",_A_NORMAL, &fh1) != 0)
  perror("Couldn't create data file");
 else
  printf("Created data file.\n");
 /* if _dos_creat is successful, the
    _dos_creatnew will fail since the file exists
 */
 if (_dos_creatnew("data",_A_RDONLY, &fh2) != 0)
  perror("Couldn't create data file");
 else
  printf("Created data file.\n");

}
```

This program creates a file using the **_dos_creat** function. The program cannot create a new file using the **_dos_creatnew** function because it already exists.

_dos_findfirst, _dos_findnext

Summary

```
#include <dos.h>

unsigned _dos_findfirst(path, attributes, buffer);

unsigned _dos_findnext(buffer);

char *path;                     Target file name
unsigned attributes;            Target attributes
struct find_t {              File-information return structure:
        char reserved[21];      Reserved for use by MS-DOS
        char attrib;            Attribute byte for matched path
        unsigned wr_time;       Time of last write to file
        unsigned wr_date;       Date of last write to file
        long size;              Length of file in bytes
        char name[13];          Null-terminated name of matched
                                file/directory, without the path
        } *buffer;
```

Description

The **_dos_findfirst** routine uses system call 0x4E to return information about the first instance of a file whose name and attributes match the *path* and *attributes* arguments. Information is returned in a **find_t** structure, defined in **dos.h**.

The *path* argument may use wildcards (* and ?). The *attributes* argument can be any of the following manifest constants:

Constant	Meaning
_A_NORMAL	Normal. File can be read or written without restriction.
_A_RDONLY	Read only. File cannot be opened for a "write," and a file with the same name cannot be created.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_A_SYSTEM	System file. Cannot be found by a directory search.
_A_VOLID	Volume ID. Only one file can have this attribute, and it must be in the root directory.

_A_SUBDIR	Subdirectory.
_A_ARCH	Archive. Set whenever the file is changed, and cleared by the MS-DOS **BACKUP** command.

Multiple constants can be ORed together, using the vertical-bar (¦) character.

The **_dos_findnext** routine uses system call 0x4F to find the next name, if any, that matches the *path* and *attributes* arguments specified in a prior call to **_dos_findfirst**. The *buffer* argument must point to a structure already initialized by a previous call to **_dos_findfirst**. The contents of the structure will be altered as described above if a match is found.

Return Value

If successful, both functions return 0. Otherwise, they return the MS-DOS error code and set **errno** to **ENOENT**, indicating that the path could not be matched.

Example

```
#include <dos.h>

main()
 {
   struct find_t  c_file;

   /* find first .c file in current directory */
   _dos_findfirst ("*.c", _A_NORMAL, &c_file);
   printf ("Listing of .c files\n\n");
   printf ("file %s is %d bytes long\n",c_file.name,
           c_file.size);
   /* find the rest of the .c files */
   while (_dos_findnext(&c_file) == 0)
      printf ("file %s is %d bytes long\n",c_file.name,
              c_file.size);
 }
```

This program finds and prints all files in the current directory with the `.c` extension.

_dos_freemem

Summary

#include <dos.h>

unsigned _dos_freemem(*segment***);**
unsigned *segment***;** Block to be released

Description

The **_dos_freemem** function uses system call 0x49 to release a block of memory previously allocated by **_dos_allocmem**. The *segment* argument is a value returned by a previous **_dos_allocmem** or **_dos_setblock** call. The freed memory may no longer be used by the application program.

Return Value

If successful, **_dos_freemem** returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to **ENOMEM**, indicating a bad segment value (one that does not correspond to a segment returned by a previous **_dos_allocmem** or **_dos_setblock** call) or invalid arena headers.

See Also

_dos_allocmem, **_dos_setblock**, **ffree**, **free**, **hfree**, **nfree**

Example

```
#include <dos.h>

unsigned segment;

main()
  {
  /* Allocate 20 paragraphs */
  if (_dos_allocmem (20, &segment) != 0)
     printf ("allocation failed\n");
  else
     printf ("allocation successful\n");
  if (_dos_freemem (segment) != 0)
     printf ("free memory failed\n");
  else
     printf ("free memory successful\n");
 }
```

This program allocates and then frees 20 paragraphs of memory space.

Summary

```
#include <dos.h>

void _dos_getdate(date);
struct dosdate_t {           Current date structure:
        unsigned char day;              1-31
        unsigned char month;            1-12
        unsigned int year;              1980-2099
        unsigned char dayofweek;             0-6 (0 = Sunday)
        } *date;
```

Description

The **_dos_getdate** routine uses system call 0x2A to obtain the current system date. The date is returned in a **dosdate_t** structure, defined in **dos.h**.

See Also

_dos_gettime, **_dos_setdate**, **_dos_settime**, **gmtime**, **localtime**, **mktime**, **_strdate**, **_strtime**, **time**

Example

```
#include <dos.h>

main()
 {
   struct dosdate_t date;
   struct dostime_t time;

  /* get current date and time values */

  _dos_getdate (&date);
  _dos_gettime (&time);
  printf("Today's date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The time is %d:%d\n",time.hour,time.minute);
 }
```

This program gets and displays the current date and time values.

_dos_getdiskfree

Summary

```
#include <dos.h>

unsigned _dos_getdiskfree(drive, diskspace);
unsigned drive;
struct diskfree_t {
        unsigned total_clusters;
        unsigned avail_clusters;
        unsigned sectors_per_cluster;
        unsigned bytes_per_sector;
        } *diskspace;
```

Description

The **_dos_getdiskfree** routine uses system call 0x36 to obtain information on the disk drive specified by *drive.* The default drive is 0, drive A is 1, drive B is 2, and so on. Information is returned in the **diskfree_t** structure that *diskspace* points to, defined in **dos.h**.

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets **errno** to **EINVAL**, indicating an invalid drive was specified.

See Also

_dos_getdrive, **_dos_setdrive**

Example

```
#include <dos.h>

main()
 {
   struct diskfree_t drive;

  /* get information on default disk drive 0 */

  _dos_getdiskfree (0, &drive);
  printf("total clusters: %d\n",drive.total_clusters);
  printf("available clusters: %d\n",drive.avail_clusters);
  printf("sectors per cluster: %d\n",drive.sectors_per_cluster);
  printf("bytes per sector: %d\n",drive.bytes_per_sector);
 }
```

This program displays information about the default disk drive.

_dos_getdrive

Summary

#include <dos.h>

void _dos_getdrive(*drive*);
**unsigned ****drive*;   Current-drive return buffer

Description

The **_dos_getdrive** routine uses system call 0x19 to obtain the current disk drive. The current drive is returned in the word that *drive* points to: 1 = drive A, 2 = drive B, and so on.

See Also

_dos_getdiskfree, **_dos_setdrive**

Example

```
#include <dos.h>

main()
 {
   unsigned drive;
   unsigned number_of_drives;

  /* print current default drive information */
  _dos_getdrive (&drive);
  printf("The current drive is: %c\n",'A' + drive - 1);

  /* set default drive to be drive A */
  _dos_setdrive (1, &number_of_drives);

  /* get new default drive information and
     total number of drives in system */
  _dos_getdrive (&drive);
  printf("The current drive is: %c\n",'A' + drive - 1);
  printf ("number of disk drives: %d\n", number_of_drives);
 }
```

This program prints the letter of the current drive, changes the default drive to A, then returns the number of disk drives.

■ Summary

#include <dos.h>

unsigned _dos_getfileattr(*path*, *attribute*);	
char **path*;	Full path of target file/directory
unsigned **attributes*;	Word to store attributes in

■ Description

The **_dos_getfileattr** routine uses system call 0x43 to obtain the current attributes of the file or directory that *path* points to. The attributes are copied to the low-order byte of the *attributes* word. Attributes are represented by manifest constants, as described below:

Constant	Meaning
_A_NORMAL	Normal. File can be read or written without restriction.
_A_RDONLY	Read only. File cannot be opened for a "write," and a file with the same name cannot be created.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_A_SYSTEM	System file. Cannot be found by a directory search.
_A_VOLID	Volume ID. Only one file can have this attribute, and it must be in the root directory.
_A_SUBDIR	Subdirectory.
_A_ARCH	Archive. Set whenever the file is changed, or cleared by the MS-DOS **BACKUP** command.

■ Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and set **errno** to **ENOENT**, indicating that the target file or directory could be found.

■ See Also

_dos_setfileattr

Example

```
#include <dos.h>

main()
   {
   unsigned attribute;
   int fh;

   /* create file as read only */
   if (_dos_creat("data",_A_RDONLY, &fh) != 0)
      perror("Couldn't create data file");
   else
      printf("Created data file.\n");

   /* get and print file attribute */
   _dos_getfileattr("data",&attribute);
   printf ("attribute: %d\n", attribute);
   if ((attribute & _A_RDONLY) != 0)
      printf("Read only file\n");
   else
      printf("Not a read only file.\n");

   /* reset file attribute to normal file */
   _dos_setfileattr("data",_A_NORMAL);
   _dos_getfileattr("data",&attribute);
   printf ("attribute: %d\n", attribute);
   if ((attribute & _A_RDONLY) != 0)
      printf("Read only file\n");
   else
      printf("Not a read only file.\n");
}
```

This program creates a file with the specified attributes then prints this information before changing the file attributes back to normal.

Summary

#include <dos.h>

unsigned _dos_getftime(*handle, date, time***);**

int *handle***;**	Target file
unsigned ****date*;**	Date-return buffer
unsigned ****time*;**	Time-return buffer

Description

The **_dos_getftime** routine uses system call 0x57 to get the date and time that the file identified by *handle* was last written. The date and time are returned in the words that *date* and *time*, respectively, point to. The values appear in the MS-DOS date and time format, which is:

Time Bits	Meaning
0-4	Seconds/2 (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)

Date Bits	Meaning
0-4	Day (1-31)
5-8	Month (1-12)
9-15	Year (1980-2099)

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to **EBADF**, indicating that an invalid file handle was passed.

See Also

_dos_setftime

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main()
 {

  unsigned date;
  unsigned time;
  int fh;

  /* open file with _dos_open function */
  if (_dos_open("dgftime.c",O_RDONLY, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* modify file date and time */
  _dos_getftime (fh, &date, &time);
  printf ("date and time read\n");
  printf ("date field (hex): %x\n", date);
  printf ("time field (hex): %x\n", time);

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program displays the date and time fields for a file.

_dos_gettime

Summary

```
#include <dos.h>

void _dos_gettime(time);
struct dostime_t {          Current system time:
        unsigned char hour;            0-23
        unsigned char minute;    0-59
        unsigned char second;    0-59
        unsigned char hsecond;   1/100 second; 0-99
        } *time;
```

Description

The **_dos_gettime** routine uses system call 0x2C to obtain the current system time. The time is returned in a **dostime_t** structure, defined in **dos.h**.

See Also

_dos_getdate, **_dos_setdate**, **_dos_settime**

Example

```
#include <dos.h>

main()
 {
   struct dosdate_t date;
   struct dostime_t time;

  /* get current date and time values */

  _dos_getdate (&date);
  _dos_gettime (&time);
  printf("Today's date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The time is %d:%d\n",time.hour,time.minute);
 }
```

This program displays the current date and time values.

_dos_getvect

Summary

#include <dos.h>

void (interrupt far *_dos_getvect(*intnum*))();
unsigned *intnum*; Target interrupt vector

Description

The **_dos_getvect** routine uses system call 0x35 to get the current value of the interrupt vector specified by *intnum*.

Return Value

The function returns a far pointer to the current handler, if any, for the *intnum* interrupt.

See Also

_chain_intr, **_dos_setvect**

Summary

#include <dos.h>

void _dos_keep(*retcode, memsize***);**	
unsigned *retcode***;**	Exit status code
unsigned *memsize***;**	Allocated resident memory (in 16-byte paragraphs)

Description

The **_dos_keep** routine installs terminate-and-stay-resident programs (TSR's) in memory, using system call 0x31. It first exits the calling process, leaving it in memory, and returns the low-order byte of *retcode* to the parent of the calling process. Before returning execution to the parent process, **_dos_keep** sets the allocated memory for the now-resident process to *memsize* paragraphs (a paragraph is 16 bytes). Any excess memory is returned to the system.

See Also

_chain_intr, **_dos_getvect**, **_dos_setvect**

_dos_open

Summary

#include <dos.h>

unsigned _dos_open(*path*, *mode*, *handle***);**	
char **path***;**	Path to an existing file
unsigned *mode***;**	Permissions
int **handle***;**	Handle return buffer

Description

The **_dos_open** routine uses system call 0x3D to open the existing file that *path* points to. The *mode* argument specifies the file's access, sharing, and inheritance modes by ORing together manifest constants from the three groups shown below. At most, one access mode and one sharing mode may be specified at a time.

Constant	Mode	Meaning
O_RDONLY	Access	Read only
O_WRONLY	Access	Write only
O_RDWR	Access	Both read and write
SH_COMPAT	Sharing	Compatibility
SH_DENYRW	Sharing	Deny reading and writing
SH_DENYWR	Sharing	Deny writing
SH_DENYRD	Sharing	Deny reading
SH_DENYNONE	Sharing	Deny neither
O_NOINHERIT	Inheritance	File is not inherited by the child process

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to one of the following manifest constants:

Constant	Meaning
EINVAL	Sharing mode specified when file sharing not installed, or access-mode value is invalid
ENOENT	Path or file not found

EMFILE	Too many open file handles
EACCES	Access denied (*path* specifies a directory or a volume ID, or opening read-only for write access)

■ See Also

_dos_close, **_dos_read**, **_dos_write**

■ Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main()
 {
  int fh;

  /* open file with _dos_open function */
  if (_dos_open("data1",O_RDONLY, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program uses the MS-DOS I/O operations to open and close a file.

_dos_read

Summary

#include <dos.h>

int _dos_read(*handle, buffer, count, bytes***);**	
int *handle***;**	File to read
void far **buffer***;**	Buffer to write to
unsigned *count***;**	Number of bytes to read
unsigned **bytes***;**	Number of bytes actually read

Description

The **_dos_read** routine uses system call 0x3F to read *count* bytes of data from the file specified by *handle* and copy it to the buffer that *buffer* points to. The integer that *bytes* points to will show the number of bytes actually read, which may be less than the number requested in *count*. If the number of bytes actually read is 0, it means the routine tried to read at EOF.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to one of the following constants:

Constant	Meaning
EBADF	Invalid file handle
EACCES	Access denied (*handle* is not open for read access)

See Also

_dos_close, _dos_open, _dos_write

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main()
 {
  int fh;
  char buffer[50];
  unsigned number_read;

  /* open file with _dos_open function */
  if (_dos_open("dread.c",O_RDONLY, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* read data with _dos_read function */
  _dos_read (fh, buffer, 50, &number_read);
  printf ("buffer contents: %s\n", buffer);

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program uses the MS-DOS I/O operations to read the contents of a file.

_dos_setblock

Summary

# include <dos.h>

unsigned _dos_setblock(*size*, *segment*, *maxsize***);**

unsigned *size*;	New segment size
unsigned *segment*;	Target segment
unsigned **maxsize*;	Maximum size buffer

Description

The **_dos_setblock** routine changes the size of *segment*, previously allocated by **_dos_allocmem**, to *size* paragraphs, using system call 0x4A. If the request canot be satisfied, the maximum possible segment size is copied to the buffer that *maxsize* points to.

Return Value

The function returns 0 if successful, or an MS-DOS error code otherwise.

Return Value

The function returns 0 if successful. If the call fails, it returns the MS-DOS error code and sets **errno** to **ENOMEM**, indicating a bad segment value was passed (one that does not correspond to a segment returned from a previous **_dos_allocmem** call) or invalid arena headers.

See Also

_dos_allocmem, **_dos_freemem**, **realloc**

Example

```
#include <dos.h>

unsigned segment;
unsigned maxsize;

main()
  {
  /* Allocate 20 paragraphs */
  if (_dos_allocmem (20, &segment) != 0)
     printf ("allocation failed\n");
  else
     printf ("allocation successful\n");

  /* Increase allocation to 40 paragraphs */
  if (_dos_setblock (40, segment, &maxsize) != 0)
     printf ("allocation increase failed\n");
  else
     printf ("allocation increase successful\n");

  /* free memory */
  if (_dos_freemem (segment) != 0)
     printf ("free memory failed\n");
  else
     printf ("free memory successful\n");
 }
```

This program allocates 20 paragraphs of memory, increases the allocation to 40 paragraphs and then frees the memory space.

_dos_setdate

Summary

```
#include <dos.h>

unsigned _dos_setdate(date);
struct dosdate_t {                    New date:
        unsigned char day;                1-31
        unsigned char month;              1-12
        unsigned int year;                1980-2099
        unsigned char dayofweek;          0-6 (0 = Sunday)
        } date;
```

Description

The **_dos_setdate** routine uses system call 0x2B to set the current system date. The date is stored in the **dosdate_t** structure that *date* points to, defined in **dos.h**.

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets **errno** to **EINVAL**, indicating an invalid date was specified.

See Also

_dos_gettime, **_dos_setdate**, **_dos_settime**, **gmtime**, **localtime**, **mktime**, **_strdate**, **_strtime**, **time**

Example

```
#include <dos.h>

main()
 {
   struct dosdate_t date;
   struct dostime_t time;

  /* get current date and time values */

  _dos_getdate (&date);
  _dos_gettime (&time);
  printf("Today's date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The time is %d:%d\n",time.hour,time.minute);
  /* set year to 1999 and the hour to 11 */
  date.year = 1999;
  time.hour = 11;

  /* modify date and time structures */
  _dos_setdate (&date);
  _dos_settime (&time);

  /* print new dates and times */
  printf("The new date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The new time is %d:%d\n",time.hour,time.minute);
 }
```

This program changes the time and date values and displays the new date and time values.

_dos_setdrive

Summary

#include <dos.h>

void _dos_setdrive(*drivenum, drives*);	
unsigned *drivenum*;	New default drive
unsigned **drives*;	Total drives available

Description

The **_dos_setdrive** routine uses system call 0x0E to set the current default drive to the *drivenum* argument: 1 = drive A, 2 = drive B, and so on. The

drives argument indicates the total number of drives in the system. If this value is 4, for example, it doesn't mean they are designated A, B, C, and D; it only means that four drives are in the system.

There is no return value. If an invalid drive number is passed, the function fails without indication. Use the **_dos_getdrive** routine to verify whether the desired drive has been set.

See Also

_dos_getdiskfree, **_dos_getdrive**

■ Example

```
#include <dos.h>

main()
 {
   unsigned drive;
   unsigned number_of_drives;

  /* print current default drive information */
  _dos_getdrive (&drive);
  printf("The current drive is: %c\n",'A' + drive - 1);

  /* set default drive to be drive A */
  _dos_setdrive (1, &number_of_drives);

  /* get new default drive information and
     total number of drives in system */
  _dos_getdrive (&drive);
  printf("The current drive is: %c\n",'A' + drive - 1);
  printf ("number of disk drives: %d\n", number_of_drives);
 }
```

This program prints the letter of the current drive, changes the default drive to A, then returns the number of disk drives.

_dos_setfileattr

Summary

#include <dos.h>

unsigned _dos_setfileattr(*path, attributes***);**	
char ****path*;**	Full path of target file/directory
unsigned *attributes***;**	New attributes

Description

The **_dos_setfileattr** routine uses system call 0x43 to set the attributes of the file or directory that *path* points to. The actual attributes are contained in the low-order byte of the *attribute* word. Attributes are represented by manifest constants, as described below:

Constant	Meaning
_A_NORMAL	Normal. File can be read or written to without restriction.
_A_RDONLY	Read only. File cannot be opened for a "write," and a file with the same name cannot be created.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_A_SYSTEM	System file. Cannot be found by a directory search.
_A_VOLID	Volume ID. Only one file can have this attribute, and it must be in the root directory.
_A_SUBDIR	Subdirectory.
_A_ARCH	Archive. Set whenever the file is changed, or cleared by the MS-DOS **BACKUP** command.

Return Value

The function returns 0 if successful. Otherwise, it returns the MS-DOS error code and sets **errno** to one of the following:

Constant	Meaning
ENOENT	No file or directory matching the target was found.
EACCES	Access denied; cannot change the volume ID or the subdirectory.

■ See Also

_dos_getfileattr

■ Example

```
#include <dos.h>

main()
   {
   unsigned attribute;
   int fh;

   /* create file as read only */
   if (_dos_creat("data",_A_RDONLY, &fh) != 0)
      perror("Couldn't create data file");
   else
      printf("Created data file.\n");

   /* get and print file attribute */
   _dos_getfileattr("data",&attribute);
   printf ("attribute: %d\n", attribute);
   if ((attribute & _A_RDONLY) != 0)
      printf("Read only file\n");
   else
      printf("Not a read only file.\n");

   /* reset file attribute to normal file */
   _dos_setfileattr("data",_A_NORMAL);
   _dos_getfileattr("data",&attribute);
   printf ("attribute: %d\n", attribute);
   if ((attribute & _A_RDONLY) != 0)
      printf("Read only file\n");
   else
      printf("Not a read only file.\n");
}
```

This program creates a file with the specified attributes, then prints a message describing these attributes, then changes the file attributes back to normal.

_dos_setftime

Summary

#include <dos.h>

unsigned _dos_setftime(*handle*, *date*, *time*);	
int *handle*;	Target file
unsigned *date*;	Date of last write
unsigned *time*;	Time of last write

Description

The **_dos_setftime** routine uses system call 0x57 to set the *date* and *time* at which the file identified by *handle* was last written to. Those values appear in the MS-DOS date and time format, which is:

Time Bits	Meaning
0-4	Seconds/2 (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)

Date Bits	Meaning
0-4	Day (1-31)
5-8	Month (1-12)
9-15	Year (1980-2099)

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to **EBADF**, indicating that an invalid file handle was passed.

See Also

_dos_getftime

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main()
 {

  unsigned date = 0x421;
  unsigned time = 0xC0F;
  int fh;

  /* open file with _dos_open function */
  if (_dos_open("dsfilt.c",O_RDONLY, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* modify file date and time */
  _dos_setftime (fh, date, time);
  printf ("date and time changed\n");

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program changes the date and time fields for a file.

_dos_settime

Summary

#include <dos.h>

```
unsigned _dos_settime(time);
struct dostime_t {          New time:
      unsigned char hour;           0-23
      unsigned char minute;    0-59
      unsigned char second;    0-59
      unsigned char hsecond;   Hundredths of a second; 0-99
      } *time;
```

Description

The **_dos_settime** routine uses system call 0x2D to set the current time to the value stored in the **dostime_t** structure that *time* points to, as defined in **dos.h**.

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets **errno** to **EINVAL**, indicating an invalid time was specified.

See Also

_dos_getdate, **_dos_gettime**, **_dos_setdate**, **gmtime**, **localtime**, **mktime**, **_strdate**, **_strtime**

Example

```
#include <dos.h>

main()
 {
   struct dosdate_t date;
   struct dostime_t time;

  /* get current date and time values */

  _dos_getdate (&date);
  _dos_gettime (&time);
  printf("Today's date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The time is %d:%d\n",time.hour,time.minute);
  /* set year to 1999 and the hour to 11 */
  date.year = 1999;
  time.hour = 11;

  /* modify date and time structures */
  _dos_setdate (&date);
  _dos_settime (&time);

  /* print new dates and times */
  printf("The new date is %d-%d-%d\n",date.month,date.day,
         date.year);
  printf("The new time is %d:%d\n",time.hour,time.minute);
 }
```

This program changes the time and date values.

_dos_setvect

Summary

#include <dos.h>

void _dos_setvect(*intnum*, **void(***handler***)());**

unsigned *intnum*;	Target interrupt vector
interrupt far **handler*;	Interrupt handler to assign *intnum* to

Description

The **_dos_setvect** routine uses system call 0x25 to set the current value of the interrupt vector *intnum* to the function that *handler* points to. Subsequently, whenever the *intnum* interrupt is generated, the *handler* routine will be called. If *handler* is a C function, it must have been previously declared with the interrupt attribute. Otherwise, you must make sure that the function satisfies the requirements for an interrupt-handling routine.

See Also

_chain_intr, **_dos_getvect**, **_dos_keep**

Summary

#include <dos.h>

unsigned _dos_write(*handle, buffer, count, bytes*);	
int *handle*;	File to write to
void far **buffer*;	Buffer to write from
unsigned *count*;	Number of bytes to write
unsigned **bytes*;	Number of bytes actually written

Description

The **_dos_write** routine uses system call 0x40 to write into the file that *handle* references *count* bytes of data from the buffer to which *buffer* points. The integer that *bytes* points to will be the number of bytes actually written, which may be less than the number requested.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets **errno** to one of the following manifest constants:

Constant	Meaning
EBADF	Invalid file handle
EACCES	Access denied (*handle* references a file not open for write access)

See Also

_dos_close, **_dos_open**, **_dos_read**

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

char out_buffer [10] = "hello";

main()
 {
  int fh;
  char in_buffer[10];
  unsigned n_read, n_written;

  /* open file with _dos_open function */
  if (_dos_open("data1",O_RDWR, &fh) != 0)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  /* write data with _dos_write function */
  _dos_write (fh, out_buffer, 10, &n_written);
  printf ("number of characters written: %d\n", n_written);

  /* close file with _dos_close function */
  if (_dos_close(fh) != 0)
    perror("close failed");
  else
    printf("file successfully closed\n");
 }
```

This program uses the MS-DOS I/O operations to write information to a file.

■ Summary

#include <dos.h>

int dosexterr (*buffer***);**
struct DOSERROR {

int exterror;	**AX** register contents
char class;	**BH** register contents
char action;	**BL** register contents
char locus;	**CH** register contents

} ***buffer*;**

■ Description

The **dosexterr** function obtains the register values returned by the MS-DOS system call 0x59 and stores the values in the structure that *buffer* points to. This function is useful when making system calls under MS-DOS versions 3.0 or later, which offer extended error handling.

The structure type **DOSERROR** is defined in **dos.h**. Giving a **NULL** pointer argument causes **dosexterr** to return the value in **AX** without filling in the structure fields. See the *MS-DOS Programmer's Reference* for more information on the register contents.

■ Return Value

The **dosexterr** function returns the value in the **AX** register (identical to the value in the **exterror** structure field).

■ See Also

perror

Note

The **dosexterr** function should be used only under MS-DOS versions 3.0 or later.

Example

```
#include <dos.h>
#include <fcntl.h>
#include <stdio.h>

struct DOSERROR doserror;
int fd;

main()
 {
  if ((fd = open("test.dat", O_RDONLY)) == -1)
   {
    dosexterr( &doserror );
    printf("error=%d, class=%d, action=%d, locus=%d\n",
            doserror.exterror, doserror.class, doserror.action,
            doserror.locus);
   }
  else
    printf("Open succeeded so no extended information printed");
 }
```

This program tries to open the file `test.dat`. If the attempted open operation fails, the program uses **dosexterr** to display extended error information.

Summary

# include <io.h>	Required only for function declarations
int dup(*handle*);	Creates second handle for open file
int *handle*;	Handle referring to open file
int dup2(*handle1*, *handle2*);	Assigns *handle2* to *handle1*'s file
int *handle1*;	Handle referring to open file
int *handle2*;	Any handle value

Description

The **dup** and **dup2** functions cause a second file handle to be associated with a currently open file. Operations on the file can be carried out using either file handle, since all handles associated with a given file use the same file pointer. The type of access allowed for the file is unaffected by the creation of a new handle.

The **dup** function returns the next available file handle for the given file. The **dup2** function forces *handle2* to refer to the same file as *handle1*. If *handle2* is associated with an open file at the time of the call, that file is closed.

Return Value

The **dup** function returns a new file handle. The **dup2** function returns 0 to indicate success. Both functions return –1 if an error occurs and set **errno** to one of the following values:

Value	Meaning
EBADF	Invalid file handle
EMFILE	No more file handles available (too many open files)

See Also

close, **creat**, **open**

Example

```
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

int old;
FILE *new;

main()
 {
  old = dup(1);       /* "old" now refers to "stdout" */
                      /* Note:  file handle 1 == "stdout" */
  if (old == -1)
  {
   perror("dup(1) failure");
   exit(1);
  }
  write(old, "This goes to stdout first\n", 27);
  if ((new = fopen("data", "w")) == NULL)
   {
    puts("Can't open file \"data\"\n");
    exit(1);
   }
   /* stdout now refers to file "data" */
  if (-1 == dup2(fileno(new), 1))
   {
    perror("Can't dup2 stdout");
    exit(1);
   }
  puts( "This goes to file \"data\"\n" );
  fflush(stdout);         /* Flush stdout stream buffer so
                          it goes to correct file */
  fclose(new);
  dup2(old, 1);           /* Restore original stdout */
  puts( "This goes to stdout" );
 }
```

This program uses the variable `old` to save the original **stdout**. It then opens a new file named new and forces **stdout** to refer to it. Finally, it restores **stdout** to its original state.

Summary

# include <stdlib.h>	Required only for function declarations
char *ecvt(*value, count, dec, sign***);**	
double *value*;	Number to be converted
int *count*;	Number of digits stored
int **dec*;	Stored decimal point position
int **sign*;	Sign of converted number

Description

The **ecvt** function converts a floating-point number to a character string. The *value* is the floating-point number to be converted. The **ecvt** function stores up to *count* digits of *value* as a string and appends a null character (**'\0'**). If the number of digits in *value* exceeds *count*, the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The argument *dec* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The argument *sign* points to an integer indicating the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

Return Value

The **ecvt** function returns a pointer to the string of digits. There is no error return.

See Also

atof, **atoi**, **atol**, **fcvt**, **gcvt**

Note

The **ecvt** and **fcvt** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

Example

```
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

main()

 {
 /* buffer will contain "3141592654"
 ** decimal = 1, sign = 0
 */
  buffer = ecvt(3.1415926535,precision,&decimal,&sign);
  printf("buffer= \"%s\", decimal = %d, sign = %d\n", \
        buffer, decimal, sign);
 }
```

This program uses **ecvt** to convert the constant 3.141592654 from a floating-point number to a character string. It then displays the resulting string.

Summary

#include <graph.h>

short far _ellipse(*control***,** *x1***,** *y1***,** *x2***,** *y2***);**	
short *control***;**	Fill flag
short *x1***,** *y1***;**	Upper-left corner of bounding rectangle
short *x2***,** *y2***;**	Lower-right corner of bounding rectangle

Description

The **_ellipse** function draws an ellipse. The border is drawn in the current color. The center of the ellipse is the center of the bounding rectangle defined by the logical points (*x1*, *y1*) and (*x2*, *y2*).

The *control* argument can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the ellipse using the current fill mask
_GBORDER	Does not fill the ellipse

If the bounding-rectangle arguments define a point or a vertical or horizontal line (*x1* = *x2* or *y1* = *y2*), no figure is drawn.

Return Value

The **_ellipse** function returns a nonzero value if the ellipse is drawn successfully; otherwise, it returns 0.

See Also

_arc, **_lineto**, **_pie**, **_rectangle**, **_setcolor**, **_setfillmask**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 _setvideomode( _MRES16COLOR );
 _ellipse( _GFILLINTERIOR, 80, 50, 240, 150 );
 while ( !kbhit()); /* Strike any key to clear screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws the shape shown in Figure R.2.

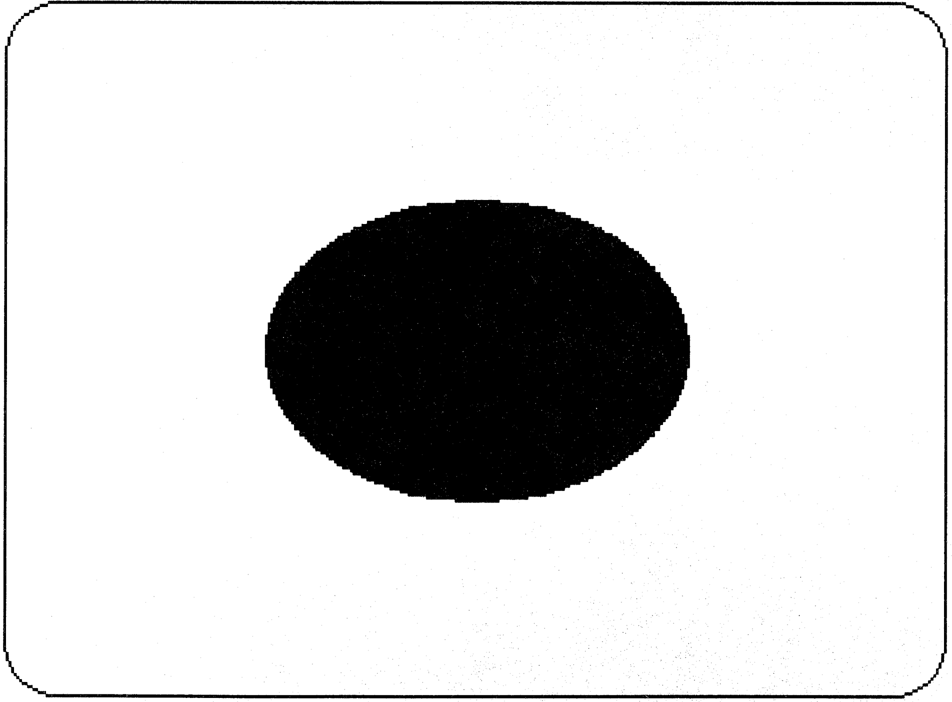


Figure R.2 Output of _ellipse Program

Summary

#include <dos.h>

void _enable(void);	Enables interrupts

Description

The **_enable** routine enables interrupts by executing an 8086 STI machine instruction.

See Also

_disable

eof

Summary

#include <io.h>	Required only for function declarations
int eof(*handle***);**	
int *handle***;**	Handle referring to open file

Description

The **eof** function determines whether the end-of-file has been reached for the file associated with *handle.*

Return Value

The **eof** function returns the value 1 if the current position is end-of-file, 0 if it is not. A return value of –1 indicates an error; in this case, **errno** is set to **EBADF**, indicating an invalid file handle.

See Also

clearerr, **feof**, **ferror**, **perror**

Example

```
#include <io.h>
#include <fcntl.h>

int fh, count;
char buf[10];

main()
 {
  int total = 0;

  fh = open("data",O_RDONLY);
  /* Cycle until end of file reached: */
  while (!eof(fh))
   {

   /* Attempt to read in 10 bytes: */
    if ((count = read(fh, buf, 10)) == -1){
      perror("Read error");
      break;
    }
    total += count;     /* Total up actual bytes read */
   }
  printf("Number of bytes read = %d\n", total);
 }
```

This program opens a file named `data` and reads data from the file until the end of the file is reached. It then uses the function named `eof` to determine when the end of the file was found. If the **read** function reports an error, reading is terminated and the current total is reported.

execl – execvpe

Summary

#include <process.h>	Required only for function declarations

int execl(*path, arg0, arg1,... argn,* **NULL);**

int execle(*path, arg0, arg1,... argn,* **NULL,** *envp***);**

int execlp(*path, arg0, arg1,... argn,* **NULL);**

int execlpe(*path, arg0, arg1,... argn,* **NULL,** *envp***);**

int execv(*path, argv***);**

int execve(*path, argv, envp***);**

int execvp(*path, argv***);**

int execvpe(*path, argv, envp***);**

`char *path;`	Path name of file to be executed
`char *arg0,*arg1,... *argn;`	List of pointers to arguments
`char *argv[ ];`	Array of pointers to arguments
`char *envp[ ];`	Array of pointers to environment settings

Description

The **exec** functions load and execute new child processes. When the call is successful, the child process is placed in the memory previously occupied by the calling process. Sufficient memory must be available for loading and executing the child process.

All of the functions in this family use the same **exec** function; the letter(s) at the end determine the specific variation:

Letter	Variation
p	Uses the **PATH** environment variable to find the file to be executed
l	Lists command-line arguments separately

v	Passes to the child process an array of pointers to command-line arguments
e	Passes to the child process an array of pointers to environment arguments

The *path* argument specifies the file to be executed as the child process. It can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *path* does not have a file-name extension or does not end with a period (.), the **exec** function searches for the file; if unsuccessful, it tries the extension **.COM**, then **.EXE**. If *path* has an extension, only that extension is used. If *path* ends with a period, the **exec** calls search for *path* with no extension. The **execlp**, **execlpe**, **execvp**, and **execvpe** routines search for *path* (using the same procedures) in the directories specified by the **PATH** environment variable.

Arguments are passed to the new process by giving one or more pointers to character strings as arguments in the **exec** call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the new process must not exceed 128 bytes. The terminating null character (**'\0'**) for each string is not included in the count, but space characters (inserted automatically to separate the arguments) are counted.

The argument pointers can be passed as separate arguments (**execl**, **execle**, **execlp**, and **execlpe**) or as an array of pointers (**execv**, **execve**, **execvp**, and **execvpe**). At least one argument, *arg0*, must be passed to the child process (which sees it as *argv*[**0**]). Usually, this argument is a copy of the *path* argument. (A different value will not produce an error.) Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* is not available for use in the child process. However, under MS-DOS Version 3.0 and later, the *path* is available as *arg0*.

The **execl**, **execle**, **execlp**, and **execlpe** calls are typically used when the number of arguments is known in advance. The argument *arg0* is usually a pointer to *path*. The arguments *arg1* through *argn* point to the character strings forming the new argument list. A null pointer must follow *argn* to mark the end of the argument list.

The **execv**, **execve**, **execvp**, and **execvpe** calls are useful when the number of arguments to the new process is variable. Pointers to the arguments are passed as an array, *argv*. The argument *argv*[**0**] is usually a pointer to *path*. The arguments *argv*[**1**] through *argv*[*n*] point to the character strings forming the new argument list. The argument *argv*[*n*+**1**] must be a null pointer to mark the end of the argument list.

Files that are open when an **exec** call is made remain open in the new process. In the **execl**, **execlp**, **execv**, and **execvp** calls, the child process inherits the environment of the parent. The **execle**, **execlpe**, **execve**, and **execvpe** calls allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. The argument *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

NAME= *value*

where **NAME** is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be **NULL**. When *envp* itself is **NULL**, the child process inherits the environment settings of the parent process.

Return Value

The **exec** functions do not normally return to the calling process. If an **exec** function returns, an error has occurred and the return value is –1. The **errno** variable is set to one of the following values:

Value	Meaning
E2BIG	The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K.
EACCES	The specified file has a locking or sharing violation (MS-DOS Versions 3.0 or later).
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	File or path name not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the child process; or the available memory has been corrupted; or an invalid block exists, indicating that the parent process was not allocated properly.

See Also

abort, **atexit**, **exit**, **_exit**, **onexit**, **spawn** functions, **system**

Note

The **exec** calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the **setmode** routine should be used to set the translation mode of these files to the desired mode.

Signal settings are not preserved in child processes created by calls to **exec** routines. The signal settings are reset to the default in the child process.

Example

```
#include <stdio.h>
#include <process.h>

char *my_env[] = {
                  "THIS=environment will be",
                  "PASSED=to child.exe by the",
                  "EXECLE=and",
                  "EXECLPE=and",
                  "EXECVE=and",
                  "EXECVPE=functions",
                  NULL
                  };

main(argc, argv)
int argc;
char *argv[];
 {
  char *args[4];
  int result;

  args[0] = "child";     /* Set up parameters to send */
  args[1] = "execv??";
  args[2] = "two";
  args[3] = NULL;
 switch (argv[1][0])   /* Based on first letter of argument */
  {
   case '1':
     execl   ("child.exe","child ","execl","two",NULL      );
     break;
   case '2':
     execle  ("child.exe","child","execle","two",NULL,my_env);
     break;
   case '3':
     execlp  ("child.exe","child","execlp","two",NULL       );
     break;
   case '4':
     execlpe("child.exe","child","execlpe","two",NULL,my_env);
     break;
   case '5':
     execv   ("child.exe",args                            );
     break;
   case '6':
     execve  ("child.exe",args,my_env                     );
     break;
   case '7':
     execvp  ("child.exe",args                            );
     break;
   case '8':
     execvpe("child.exe",args,my_env                      );
     break;
   default:
     printf("Enter a number from 1 to 8 as a "
            "command line parameter.");
     exit(1);
  }
     printf("Process was not spawned.\n");
     printf("Program 'child' was not found.");
}
```

This program accepts a number in the range 1 through 8 from the command line. Based on the number it receives, it executes one of the eight different procedures that spawn the process named `child`. For some of these procedures, the `child.exe` file must be in the same directory; for others, it need only be in the same path.

Summary

# include <process.h>	Required only for function declarations
# include <stdlib.h>	Use either **process.h** or **stdlib.h**
void exit(*status*);	Terminates after closing files
void _exit(*status*);	Terminates without flushing stream buffers
int *status*;	Exit status

Description

The **exit** and **_exit** functions terminate the calling process. The **exit** function first calls the functions registered by **atexit** and **onexit**, then flushes all buffers and closes all open files before terminating the process. The **_exit** function terminates the process without processing **atexit** or **onexit** functions or flushing stream buffers. The *status* value is typically set to 0 to indicate a normal exit and set to some other value to indicate an error.

Although the **exit** and **_exit** calls do not return a value, the low-order byte of *status* is made available to the waiting parent process, if there is one, after the calling process exits. The *status* value is available to the MS-DOS batch command **IF_ERRORLEVEL**.

Return Value

There is no return value.

See Also

abort, **atexit**, **exec** functions, **onexit**, **spawn** functions, **system**

Example

```
#include <stdio.h>

main()
 {
  FILE *stream;
  char aChar;

  stream = fopen("data", "w+");
  printf("About to exit...\nFlush buffers for the");
  printf(" file 'data'? (y/n): ");
  aChar = getch();
  aChar = toupper(aChar);
  fprintf(stream, "This will appear in \"data\" only if ");
  fprintf(stream, "buffers are flushed.\n");
  if (aChar == 'Y'){
     printf("\nExiting and flushing buffers");
     exit(0);
    }
  else{
    printf("\nExiting, but buffers are not flushed");
    _exit(0);
   }
 }
```

This program opens the file named `data`, then prompts the user to choose how to close the file. Based on the user's choice, the program closes the file using the **exit** function, which flushes buffers, or the **_exit** function, which does not.

Summary

#include <math.h>

double exp(*x*);
double *x*; Floating-point value

Description

The **exp** function returns the exponential function of its floating-point argument *x*.

Return Value

The **exp** function returns $\mathbf{e}^x$. The function returns **HUGE_VAL** on overflow, and sets **errno** to **ERANGE**; on underflow, **exp** returns 0, but does not set **errno**.

See Also

log

Example

```
#include <math.h>

main()
 {
  double x,y;
  x = 2.302585093;
  y = exp(x);                          /* y = 40 */
  printf("The exp(%f) = %f",x,y);
 }
```

This program displays the value of $e^{2.302585093}$.

Summary

#include <malloc.h>	Required only for function declarations
void *_expand(*block, size***);**	
void ****block*;**	Pointer to previously allocated memory block
size_t *size***;**	New size in bytes

Description

The **_expand** function changes the size of a previously allocated memory block by attempting to expand or contract the block without moving its location in the heap. The *block* argument points to the beginning of the block. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

The *block* argument can also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **_expand**, **halloc**, **malloc**, or **realloc** since the block was freed. If *block* points to a freed block, the block remains free after the call to **_expand**.

Return Value

The **_expand** function returns a **void** pointer to the reallocated memory block. Unlike **realloc**, **_expand** cannot move a block to change its size. This means the *block* argument to **_expand** is the same as the return value if there is sufficient memory available to expand the block without moving it.

The return value is **NULL** if there is insufficient memory available to expand the block to the given size without moving it. In this case, the item *block* points to will have been expanded as much as possible in its current location.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. The new size of the item can be checked with the **_msize** function. To get a pointer to a type other than **char**, use a type cast on the return value.

See Also

calloc, **free**, **halloc**, **malloc**, **_msize**, **realloc**

Example

```
#include <stdio.h>
#include <malloc.h>

main()
 {
  long *oldptr;
  size_t newsize = 64000;

  /* Get original memory: */
  oldptr = (long *)malloc(10000*sizeof(long));
  printf("Size of memory block pointed to by oldptr = %u\n",
    _msize(oldptr));

  /* Test whether _expand succeeded: */
  if (_expand(oldptr,newsize) != NULL)
    printf("Expand was able to increase block to %u\n",
      _msize(oldptr));

  /* Otherwise _expand failed: */
  else
    printf("Expand was able to increase block to only %u\n",
      _msize(oldptr));
 }
```

Sample output:

```
Size of memory block pointed to by oldptr = 40000
Expand was able to increase block to only 44718
```

This program allocates a block of memory for `oldptr` and uses **_msize** to display the size of that block. Next, it uses **expand** to expand the amount of memory used by `oldptr`. Finally, it calls **_msize** again to display the new amount of memory allocated to **oldptr**.

fabs

Summary

#include <math.h>

double fabs(*x*);
double *x*; Floating-point value

Description

The **fabs** function returns the absolute value of its floating-point argument.

Return Value

The **fabs** function returns the absolute value of its argument. There is no error return.

See Also

abs, **cabs**, **labs**

Example

```
#include <stdio.h>
#include <math.h>

main()
 {
  double x,y;

  x = -3.141593;
  y = fabs(x);                                /* y = 3.141593 */
  printf("The fabs(%f) is %f",x,y);
 }
```

This program displays the absolute value of –3.141593.

Summary

#include <stdio.h>

int fclose(*stream*);	Closes an open stream
FILE *_stream_;	Target **FILE** structure
int fcloseall(void);	Closes all open streams

Description

The **fclose** function closes the given *stream*. The **fcloseall** function closes all open streams except **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. It also closes any tempory files created by **tmpfile**. All buffers associated with the stream are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned by the user with **setbuf** and **setvbuf** are not automatically released.

Return Value

The **fclose** function returns 0 if the stream is successfully closed. The **fcloseall** function returns the total number of streams closed. Both functions return **EOF** to indicate an error.

See Also

close, **fdopen**, **fflush**, **fopen**, **freopen**

Example

```
#include <stdio.h>

FILE *stream, *stream2;

main()
 {
  int numclosed;

  /* Two files are opened: */
  stream  = fopen("data", "r" );
  stream2 = fopen("data2","w+");

  if (stream == NULL)
    printf("The file data was not opened\n");
  else
   {
    fclose(stream);
    printf("The file 'data' closed\n");
   }

  /* All other files are closed: */
  numclosed = fcloseall();

  printf("The function fcloseall closed %u files\n", numclosed);
 }
```

This program opens files named `data` and `data2`. It uses **fclose** to close `data` and **fcloseall** to close all remaining files.

Summary

#include <stdlib.h>	Required only for function declarations
char *fcvt(*value***,** *count***,** *dec***,** *sign***);**	
double *value***;**	Number to be converted
int *count***;**	Number of digits after decimal point
int ****dec*;**	Pointer to stored decimal-point position
int ****sign*;**	Pointer to stored sign indicator

Description

The **fcvt** function converts a floating-point number to a character string. The *value* is the floating-point number to be converted. The **fcvt** function stores the digits of *value* as a string and appends a null character ('**\0**'). The argument *count* specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer digits of precision than *count*, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained after the call from *dec* and *sign*. The argument *dec* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The argument *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

Return Value

The **fcvt** function returns a pointer to the string of digits. There is no error return.

See Also

atof, **atoi**, **atol**, **ecvt**, **gcvt**

Note

> The **ecvt** and **fcvt** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

■ Example

```
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

main()
 {
 /* buffer to contain "31415926535", decimal = 1, sign = 0 */
  buffer = fcvt(3.1415926535,precision,&decimal,&sign);
  printf("buffer= \"%s\", decimal = %d, sign = %d\n", buffer,
          decimal, sign);
 }
```

This program converts the constant 3.1415926535 to a string and sets the pointer `*buffer` to point to that string.

Summary

#include <stdio.h>

FILE *fdopen(*handle*, *type*);	
int *handle*;	Handle referring to open file
char **type*;	Type of access permitted

Description

The **fdopen** function associates an input/output stream with the file identified by *handle*, thus allowing a file opened for "low-level" I/O to be buffered and formatted. (See Section 4.7, "Input and Output," for an explanation of stream I/O and low-level I/O.) The *type* character string specifies the type of access requested for the file, as follows:

Type	Description
"r"	Opens for reading (the file must exist).
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending); creates the file first if it doesn't exist.
"r+"	Opens for both reading and writing. (The file must exist.)
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Opens for reading and appending; creates the file first if it doesn't exist.

Important

Use the **"w"** and **"w+"** modes with care, as they can destroy existing files.

The specified type must be compatible with the access mode and/or sharing modes with which the file was opened. It is the user's responsibility to ensure that this compatibility is maintained.

When a file is opened with "**a**" or "**a+**" type, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "**r+**", "**w+**", or "**a+**" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching from reading to writing or vice versa, there must be an intervening **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the values listed above, one of the following characters can be appended to the *type* string or inserted before the + character to specify the translation mode for new lines. For example, **r+b** is the same as **rb+**.

Mode	Meaning
t	Opens in text (translated) mode. Carriage-return–line-feed (CR-LF) combinations are translated into a single line feed (LF) on input; line-feed characters are translated to carriage-return–line-feed combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading, or for reading and writing, the run-time library checks for a CTRL+Z character and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file that ends with CTRL+Z may cause **fseek** to behave improperly near the end of the file. The **t** option is not part of the ANSI standard for **open**, but is a Microsoft extension and should not be used where ANSI portability is desired.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in the *type* string, the translation mode is defined by the default mode variable **_fmode**.

■ Return Value

The **fdopen** function returns a pointer to the open stream. A null pointer value indicates an error.

See Also

dup, **dup2**, **fclose**, **fcloseall**, **fopen**, **freopen**, **open**

Example

```
#include <stdio.h>
#include <fcntl.h>

FILE *stream;
int fh;

main()
 {
  fh = open("data",O_RDONLY);

  /* Buffer associated with "fh": */
  stream = fdopen(fh,"r");
  if (stream == NULL)
    printf( "Error in fdopen attempt.\n" );
  else
    printf( "Input buffer successfully associated with 'data'");
 }
```

This program opens a file named `data` and uses **fdopen** to associate an input stream with `data`.

feof

Summary

#include <stdio.h>

int feof(*stream*);	
FILE *_stream_;	Pointer to **FILE** structure

Description

The **feof** routine (implemented as a macro) determines whether the end of *stream* has been reached. Once end-of-file is reached, read operations return an end-of-file indicator until the stream is closed or **rewind** is called against it.

Return Value

The **feof** function returns a nonzero value after the first read operation that attempts to read past the end of the file. It returns 0 if the current position is not end-of-file. There is no error return.

See Also

clearerr, **eof**, **ferror**, **perror**

Example

```
#include <stdio.h>
#define BUF_SIZE  100

char string[BUF_SIZE];
FILE *stream;
main()
{
 stream = fopen("data", "r" );
 while (fgets(string,BUF_SIZE,stream))
  printf("%s",string);
 if (feof(stream))
  printf("EOF reached\n");
 else
  printf("Error reading stream\n");
}
```

This program uses `feof` to indicate when it reaches the end of the file `data`.

Summary

#include <stdio.h>

int ferror(*stream***);**
FILE **stream*; Pointer to **FILE** structure

Description

The **ferror** routine (implemented as a macro) tests for a reading or writing error on *stream*. If an error has occurred, the error indicator for the stream remains set until the stream is closed or rewound, or until **clearerr** is called against it.

Return Value

If no error has occurred on *stream*, **ferror** returns 0. Otherwise, it returns a nonzero value.

See Also

clearerr, **eof**, **feof**, **fopen**, **perror**

Example

```
#include <stdio.h>

FILE *stream;
char *string = "This should never be written";
main()
{
 stream = fopen("data", "r");
 fprintf(stream,"%s\n",string);
 if (ferror(stream)) {
  fprintf(stderr,"Write error\n");
  clearerr(stream);
 }
}
```

This program opens a file named `data` for reading and tries to write to it, causing an error. The program uses **ferror** to detect the error, then clears the error.

fflush

Summary

#include <stdio.h>

int fflush(*stream*);
FILE ****stream*; Pointer to **FILE** structure

Description

If *stream* is open for output, **fflush** writes to the associated file the contents of the buffer associated with the stream. If the stream is open for input, **fflush** clears the contents of the buffer. The **fflush** function negates the effect of any prior call to **ungetc** against *stream.*

The stream remains open after the call. The **fflush** function has no effect on an unbuffered stream.

Return Value

The **fflush** function returns the value 0 if the buffer was successfully flushed. The value 0 is also returned in cases where the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.

See Also

fclose, **flushall**, **setbuf**

Note

Buffers are automatically flushed when they are full, when the stream is closed, or when a program terminates normally without closing the stream.

Example

```
#include <stdio.h>
#include <process.h>

FILE *stream;
char buffer[BUFSIZ];

main()
 {
  int result;

  /* Redirect stdout to "data" */
  stream = freopen("data", "w", stdout);

  printf("This is the output of child:\n\n");

  /* Now make sure printf() output goes to
  ** "data" before child's output does:
  */
  result = fflush(stream);


  spawnl(P_WAIT, "child.exe", "child", "one", "two", NULL);

  printf("--------------------------------------------\n");

 }
```

This program first redirects **stdout** to a file named `data`. It uses **printf** to write to `data`, then uses **fflush** to guarantee that the output from **printf** is written before the output from the child process.

fgetc, fgetchar

Summary

#include <stdio.h>

int fgetc(*stream***);**	Reads a character from *stream*
FILE ****stream*;**	Pointer to **FILE** structure
int fgetchar(void);	Reads a character from **stdin**

Description

The **fgetc** function reads a single character as an **unsigned int** character converted to an **int** from the input *stream* at the current position. The function then increases the associated file pointer (if any) to point to the next character. The **fgetchar** function is equivalent to **fgetc(stdin)**.

Return Value

The **fgetc** and **fgetchar** functions return the character read. A return value of **EOF** may indicate an error or end-of-file; however, the **EOF** value is also a legitimate integer value, so **feof** or **ferror** should be used to verify any error or end-of-file condition.

See Also

fputc, **fputchar**, **getc**, **getchar**

Note

The **fgetc** and **fgetchar** routines are identical to **getc** and **getchar**, but are functions, not macros.

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;

main()
 {

  /* Open file to read line from: */
  stream = fopen("fgetc.c", "r");

  /* Read in first 80 characters and */
  /*  place them in "buffer": */

  ch = fgetc(stream);
  for(i=0;(i < 80)&&(feof(stream) == 0)&&(ch != '\n');i++){
        buffer[i]=ch;
        ch = fgetc(stream);
        }

  buffer[i] = '\0';          /* Add null to end string */
  printf( "%s\n", buffer );
 }
```

This program uses **getc** to read the first 80 input characters (or until the end of input) and place them into a string named `buffer`.

fgetpos

Summary

#include <stdio.h>

int fgetpos(*stream*, *pos***);**	
FILE **stream***;**	Target stream
fpos_t **pos***;**	Position indicator storage

Description

The **fgetpos** function gets the current value of *stream*'s file-position indicator and stores it in the object that *pos* points to. The **fsetpos** function can later use information stored in *pos* to reset *stream*'s pointer to its position at the time **fgetpos** was called.

Note

> The *pos* value is stored in an internal format and is intended for use only by the **fgetpos** and **fsetpos** functions.

Return Value

If successful, the **fgetpos** function returns 0. On failure, it returns a nonzero value and sets **errno** to one of the following manifest constants (defined in **stdio.h**):

Constant	Meaning
EINVAL	The *stream* value is invalid.
EBADF	The specified stream is not a valid file handle or is not accessible.

See Also

fsetpos

■ Example

```
#include <stdio.h>

FILE *stream;
fpos_t *pos;
int val;
char list[100];

main()
{
 stream = fopen("file1","rb");                  /* Open file1 */
 fread(list,sizeof(char),100,stream);      /* Read some data */
 if (fgetpos(stream,pos) != 0)     /* Save current position */
  perror("fgetpos error");
 fread(list,sizeof(char),100,stream);      /* Read some more */
 if (fsetpos(stream,pos) != 0)  /* Return to saved position */
  perror("fsetpos error");
}
```

This program opens a file named `file1` and reads 100 characters. It then calls **fgetpos** to find and save the file position pointer. After performing another read, the program calls **fsetpos** to restore the file pointer to the saved position.

fgets

Summary

#include <stdio.h>

char *fgets(*string, n, stream***);**	Reads a string from *stream*
char **string;*	Storage location for data
int *n;*	Number of characters stored
FILE **stream;*	Pointer to **FILE** structure

Description

The **fgets** function reads a string from the input *stream* and stores it in *string*. Characters are read from the current stream position up to and including the first new-line character (**'\n'**), up to the end of the stream, or until the number of characters read is equal to *n* –1, whichever comes first. The result is stored in *string*, and a null character (**'\0'**) is appended. The new line, if read, is included in the string. If *n* is equal to 1, *string* is empty (""). The **fgets** function is similar to the **gets** function; however, **gets** replaces the new-line character with **NULL**.

Return Value

If successful, the **fgets** function returns *string*. It returns **NULL** to indicate either an error or end-of-file condition. Use **feof** or **ferror** to determine whether an error occurred.

See Also

fputs, **gets**, **puts**

Example

```
#include <stdio.h>

FILE *stream;
char line[100], *result;
main()
{
 stream = fopen("fgets.c", "r");
 result = fgets(line,100,stream);
 printf("%s", line);
}
```

This program uses **fgets** to display a line from a file on the screen.

Summary

#include <math.h>

int fieeetomsbin(*src4*, *dst4***);**	IEEE floating point to MS binary floating point
int fmsbintoieee(*src4*, *dst4***);**	MS binary floating point to IEEE floating point

float **src4*, **dst4*;

Description

The **fieeetomsbin** routine converts a single-precision floating-point number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft binary format. The **fmsbintoieee** routine converts a floating-point number in Microsoft binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random-access data files created with Microsoft BASIC (which store floating-point numbers in the Microsoft binary format), and vice versa.

The argument *src4* points to the **float** value to be converted. The result is stored at the location given by *dst4*.

Return Value

These functions return 0 if the conversion is successful, and 1 if the conversion causes an overflow.

See Also

dieeetomsbin, **dmsbintoieee**

Note

These routines do not handle IEEE NANs and infinities. IEEE denormals are treated as 0 in the conversions.

filelength

Summary

#include <io.h>	Required only for function declarations
long filelength(*handle*);	
int *handle*;	Target file handle

Description

The **filelength** function returns the length, in bytes, of target file *handle*.

Return Value

The **filelength** function returns the file length in bytes. A return value of –1L indicates an error, and an invalid handle sets **errno** to **EBADF**.

See Also

chsize, **fileno**, **fstat**, **stat**

Example

```
#include <io.h>
#include <stdio.h>

FILE *stream;
long length;

main()
 {
  stream = fopen("data","r");
  /* Get length or -1L if function fails: */
  length = filelength(fileno(stream));
  if (length == -1L)            /* If function failed... */
    printf("filelength failed");
  else
    printf( "file length is %ld\n", length );
 }
```

This program opens a file named `data`, using **filelength** to determine its length. If **filelength** fails, it returns –1L and the program displays a failure message. Otherwise, the program displays the length of `data`.

Summary

#include <stdio.h>

int fileno(*stream*);
FILE **stream*; Pointer to **FILE** structure

Description

The **fileno** function returns the file handle currently associated with *stream*. If more than one handle is associated with the stream, the return value is the handle assigned when the stream was initially opened.

Return Value

The **fileno** function returns the file handle. There is no error return. The result is undefined if *stream* does not specify an open file.

See Also

fdopen, **filelength**, **fopen**, **freopen**

Note

The **fileno** routine is implemented as a macro.

Example

```
#include <stdio.h>

main()
 {
  int result = fileno(stderr);              /* result is 2 */
  printf("The file handle for stderr is %d\n", result);
 }
```

This program uses **fileno** to obtain the file handle of **stderr**.

_floodfill

Summary

short far _floodfill(*x*, *y*, *boundary*);

short *x*, *y*;	Start point
short *boundary*;	Fills boundary color

Description

The **_floodfill** function fills an area of the display using the current color and fill mask. Filling starts at the logical point (*x*, *y*). If this point lies inside the figure, the interior is filled; if outside the figure, the background is filled. The point must be inside or outside the figure to be filled, not on the figure boundary itself. Filling occurs in all directions, stopping at the color of *boundary*.

Return Value

The **_floodfill** function returns a nonzero value if the fill is successful. It returns 0 if the fill could not be completed, the starting point lies on the *boundary* color, or the start point lies outside the clipping region.

See Also

_getcolor, **_getfillmask**, **_setfillmask**, **_setcliprgn**, **_setcolor**

Example

```
#include <stdio.h>
#include <malloc.h>
#include <graph.h>

char far *buffer;

main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 for ( xvar = 163, loop = 0; xvar < 320; loop++, xvar += 3 ) {
  _setcolor(loop % 16 );
  yvar = xvar * 5 / 8;
  _rectangle(_GBORDER, 320-xvar, 200-yvar, xvar, yvar);
  _setcolor(rand(1) % 16 );
  _floodfill(0, 0, loop % 16 );
 }
 buffer = (char far *)malloc( (unsigned int)
          _imagesize( 0, 0, 80, 50 ) );
 if ( buffer == (char far *)NULL ) {
  exit( -1 );
 }
 _getimage(0, 0, 80, 50, buffer );
 _putimage( 80, 50, buffer, _GXOR );
 free((char *)buffer);
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a series of nested rectangles in different colors, while constantly changing the background color.

floor

Summary

#include <math.h>

double floor(*x*);	
double *x*;	Floating-point value

Description

The **floor** function returns a floating-point value representing the largest integer that is less than or equal to *x*.

Return Value

The **floor** function returns the floating-point result. There is no error return.

See Also

ceil, **fmod**

Example

```
#include <math.h>

main()
 {
  double y;
  y = floor(2.8);                           /* y is 2.0 */
  printf("The floor of 2.8 is %f\n",y);

  y = floor(-2.8);                          /* y is -3.0 */
  printf("The floor of -2.8 is %f\n",y);
 }
```

This example displays the largest integers less than or equal to the floating-point values 2.8 and –2.8.

Summary

#include <stdio.h>

int flushall(void);

Description

The **flushall** function writes the contents of all buffers associated with open output streams to their associated files. All buffers associated with open input streams are cleared of their current contents; the next read operation (if there is one) then reads new data from the input files into the buffers.

All streams remain open after the call to **flushall**.

Return Value

The **flushall** function returns the number of open streams (input and output). There is no error return.

See Also

fflush

Note

> Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

Example

```
#include <stdio.h>

main()
 {
  int numflushed;

  numflushed = flushall();
  printf("There were %d streams flushed\n", numflushed);
 }
```

This program uses **flushall** to flush all buffers, including **stdin**, **stdout**, and **stderr**, and prints the number of open streams.

Summary

#include <math.h>

double fmod(*x*, *y*);
double *x*, *y*; Floating-point values

Description

The **fmod** function calculates the floating-point remainder f of (x, y) such that $x = iy + f$, where i is an integer, f has the same sign as x, and the absolute value of f is less than the absolute value of y.

Return Value

The **fmod** function returns the floating-point remainder. If y is 0, the function returns 0.

See Also

ceil, **fabs**, **floor**

Example

```
#include <math.h>

main()
 {
  double x,y,z;

  x = -10.0;
  y = 3.0;
  z = fmod(x,y);                      /* z is -1.0 */
  printf("fmod(%.2f, %.2f) is %f",x,y,z);
 }
```

This program displays the floating-point remainder of –10/3.

fopen

Summary

#include <stdio.h>

FILE *fopen(*path*, *type*);	
const char ****path*;**	Path name of file
const char ****type*;**	Type of access permitted

Description

The **fopen** function opens the file specified by *path*. The character string *type* specifies the type of access requested for the file, as follows:

Type	Description
"r" read	Opens for reading. If **r** is the first character in *type*, and the file does not exist or cannot be found, the **fopen** call will fail.
"w" write	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a" append	Opens for writing at the end of the file (appending); creates the file first if it doesn't exist.
"r+"	Opens for both reading and writing. (The file must exist.)
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Opens for reading and appending; creates the file first if it doesn't exist.

Note

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** type, all write operations occur at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "**r+**", "**w+**", or "**a+**" type is specified, both reading and writing are allowed (the file is said to be open for “update”). However, when switching between reading and writing, there must be an intervening **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the values listed above, one of the following characters can be appended to *type* or inserted before the + character to specify the translation mode for new lines. For example, **r+b** is the same as **rb+**.

Mode	Meaning
t	Open in text (translated) mode. In this mode, carriage-return–line-feed (CR-LF) combinations are translated into single line feeds (LF) on input and LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, **fopen** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file that ends with a CTRL+Z may cause **fseek** to behave improperly near the end of the file. The **t** option is not part of the ANSI standard for **open**, but is a Microsoft extension and should not be used where ANSI portability is desired.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *type*, the translation mode is defined by the default-mode variable **_fmode**.

Return Value

The **fopen** function returns a pointer to the open file. A null pointer value indicates an error.

See Also

fclose, **fcloseall**, **fdopen**, **ferror**, **fileno**, **freopen**, **open**, **setmode**

Example

```
#include <stdio.h>

FILE *stream;

main()
 {

 /* Attempt to open the file: */
  if ((stream = fopen("data","r")) == NULL)
    printf("Could not open file\n");
  else
    printf( "File opened for reading\n" );
 }
```

Sample command line:

```
update employ.dat
```

Output:

```
C:\BIN\UPDATE.EXE couldn't open file employ.dat
```

This program uses **fopen** to open a file named `data` for input.

Summary

#include <dos.h>

unsigned FP_OFF(*address*);

unsigned FP_SEG(*address*);

char far ****address*;**	Long pointer to memory address

Description

The **FP_OFF** and **FP_SEG** macros can be used to set or get the offset and segment, respectively, of *address*. In small and medium memory models, the **FP_SEG** and **FP_OFF** macros only work if the far pointer argument lies in the default data segment. If the far pointer is itself in a far data segment, the macros will not work correctly.

Return Value

The **FP_OFF** macro returns an offset. The **FP_SEG** macro returns a segment address.

See Also

segread

Example

```
#include <dos.h>
#include <malloc.h>
#include <stdio.h>

char far *p;
unsigned int seg_val;
unsigned int off_val;

main()
 {
  p = _fmalloc(100);          /* Points pointer at something */
  seg_val = FP_SEG(p);        /* Gets address pointed to */
  off_val = FP_OFF(p);
  printf("Segment is %d; Offset is %d\n", seg_val, off_val);
 }
```

This program uses **FP_SEG** and **FP_OFF** to obtain the segment and offset of the long pointer `p`.

Summary

#include <float.h>

void _fpreset(void); Reinitializes floating-point-math package

Description

The **_fpreset** function reinitializes the floating-point-math package. This function is usually used in conjunction with **signal**, **system**, or the **exec** or **spawn** family.

If a program traps floating-point error signals (**SIGFPE**) with **signal**, it can safely recover from floating-point errors by invoking **_fpreset** and using **longjmp**.

Note

> On MS-DOS versions prior to 3.0, a child process executed by **exec**, **spawn**, or **system** may affect the floating-point state of the parent process if an 8087 or 80287 coprocessor is used. Therefore, if you are using either coprocessor, the following precautions are recommended:
>
> - The **exec**, **spawn**, and **system** functions should not be called during the evaluation of a floating-point expression.
> - The **_fpreset** function should be called after these routines if there is a possibility of the child process performing any floating-point operations.

Return Value

There is no return value.

See Also

exec functions, **signal**, **spawn** functions

Example

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <float.h>

int fphandler();
jmp_buf mark;
double a = 1.0, b = 0.0, c;

main()
 {
  /* Set up pointer to error handler: */
  if (signal(SIGFPE,fphandler) == (int(*)())-1)
    abort();
  if (setjmp(mark) == 0)          /* Save stack environment */
   {
    /* Generate divide by zero error: */
    c = a/b;
    printf("Should never get here\n");
   }
  printf("Recovered from floating-point error\n");
 }

int fphandler(sig,num)
int sig,num;
 {
  printf("signal = %d subcode = %d\n", sig, num);
  /* Initialize floating-point package: */
  _fpreset();
  /*  Restore environment; return -1: */
  (mark, -1)
 }
```

This program uses **signal** to set up a routine for handling floating-point errors. This routine, `fphandler`, displays an error message and reinitializes the floating-point-math package using **_fpreset**.

Summary

#include <stdio.h>

int fprintf(*stream, format*⟦, *argument*⟧...**);**	
FILE **stream***;**	Pointer to **FILE** structure
const char **format***;**	Format-control string

Description

The **fprintf** function formats and prints a series of characters and values to the output *stream.* Each *argument* (if any) is converted and output according to the corresponding format specification in *format.*

The format is of the same form and function as the format argument for the **printf** function; see the **printf** reference page for descriptions of *format* and *argument.*

Return Value

The **fprintf** function returns the number of characters printed.

See Also

cprintf, **fscanf**, **printf**, **sprintf**

Example

```
#include <stdio.h>
#include <process.h>

FILE *stream;
int i = 10;
double fp = 1.5;
char *s = "this is a string";
char c = '\n';

main()
{
 stream = fopen("results", "w");
 fprintf(stream, "%s%c", s, c);
 fprintf(stream, "%d\n",i);
 fprintf(stream, "%f\n", fp);        /* Print 1.500000 */
 fclose(stream);
 system("type results");
}
```

This program uses **fprintf** to format various data and print them to the file named `results`. It then displays `results` on the screen.

fputc, fputchar

Summary

# include <stdio.h>

int fputc(*c***,** *stream***);**	Writes a character to *stream*
int *c***;**	Character to be written
FILE **stream***;**	Pointer to **FILE** structure

int fputchar(*c***);**	Writes a character to **stdout**
int *c***;**	Character to be written

Description

The **fputc** function writes the single character *c* to the output *stream* at the current position. The **fputchar** function is equivalent to **fputc(***c***, stdout)**.

Return Value

The **fputc** and **fputchar** functions return the character written. A return value of **EOF** may indicate an error; however, since the **EOF** value is also a legitimate integer value, use **ferror** to verify an error condition.

Note

> The **fputc** and **fputchar** routines are similar to **putc** and **putchar**, but are functions, not macros.

See Also

fgetc, **fgetchar**, **putc**, **putchar**

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;

main()
 {
  stream = stdout;

  /* Demonstrate "fputc";
  ** Set up buffer:
  */
  strcpy(buffer, "This is a test of fputc!!\n");
  /* Print line to stream */
  for (i = 0; ( i < 81 ) &&
    ((ch = fputc(buffer[i],stream)) != EOF); i++);

  /* Demonstrate "fputchar";
  ** Set up buffer:
  */
  strcpy(buffer, "This is a test of fputchar!!");

  /* Print line to stream */
  for (i = 0; ( i < 81 ) &&
    ((ch = fputchar(buffer[i])) != EOF); i++);
 }
```

This program uses **fputc** and **fputchar** to send a character array to **stdout**.

Summary

#include <stdio.h>

int fputs(*string, stream***);**	Writes a string to *stream*
char **string*;	String to be output
FILE **stream*;	Pointer to **FILE** structure

Description

The **fputs** function copies *string* to the output *stream* at the current position. The terminating null character ('\0') is not copied.

Return Value

The **fputs** function returns a 0 is successful. If the function fails, it returns a nonzero value.

C 4.0 Difference

In Microsoft C Version 4.0, **fputs** returns the last character output or an EOF to indicate an error.

See Also

fgets, **gets**, **puts**

Example

```
#include <stdio.h>

FILE *stream;

main()
{
 int result;

 stream = stdout;
 result = fputs("Data files have been updated\n",stream);
}
```

This program uses **fputs** to write a single line to a stream.

Summary

#include <stdio.h>

size_t fread(*buffer, size, count, stream***);**	
void **buffer*;	Storage location for data
size_t *size*;	Item size in bytes
size_t *count*;	Maximum number of items to be read
FILE **stream*;	Pointer to **FILE** structure

Description

The **fread** function reads up to *count* items of *size* bytes from the input *stream* and stores them in *buffer*. The file pointer associated with *stream* (if there is one) is increased by the number of bytes actually read.

If the given stream is opened in text mode, carriage-return–line-feed (CR-LF) pairs are replaced with single line-feed (LF) characters. The replacement has no effect on the file pointer or the return value.

The file-pointer position cannot be determined if an error occurs, nor can the value of a partially-read item be determined.

Return Value

The **fread** function returns the number of full items actually read, which may be less than *count* if an error occurs or if the file end is encountered before reaching *count*.

The **feof** or **ferror** function should be used to distinguish a read error from an end-of-file condition. If *size* or *count* is 0, **fread** returns 0 and the buffer contents are unchanged.

See Also

fwrite, read

Example

```
#include <stdio.h>

FILE *stream;
long list[100];
int numread;
int numwritten;

main()
 {

   /* Open file in "binary" mode: */
   if ((stream = fopen("data", "w+b")) != NULL)
    {

     /* Write 100 long integers to "stream": */
     numwritten = fwrite((char *)list,sizeof(long),100,stream);
     printf( "Wrote %d items\n", numwritten );
    }
   else
     printf("Problem opening the file");

   fclose(stream);

   if ((stream = fopen("data", "r+b")) != NULL)
    {

     /* Attempt to read in 100 long ints: */
     numread = fread((char *)list,sizeof(long),100,stream);
     printf("Number of items read = %d\n", numread);
    }
   else
     printf("Was not able to open the file");
 }
```

This program opens a file named `data.bin` and writes 100 long integers to the file. It then tries to open `data.bin` and read in 100 long integers. If the attempt succeeds, the program displays the number of actual items read.

Summary

#include <stdlib.h>	For ANSI compatibility (**free** only)
#include <malloc.h>	Required only for function declarations
void free(*buffer***);**	Frees memory block
void ****buffer*;**	Allocated memory block
void _ffree(*buffer***);**	Frees block in far heap
void far ****buffer*;**	Allocated memory block in far heap
void _nfree(*buffer***);**	Frees block in near heap
void near ****buffer*;**	Allocated memory block in near heap

Description

The **free** function deallocates a memory block. The argument *buffer* points to a memory block previously allocated through a call to **calloc**, **malloc**, or **realloc**. The number of bytes freed is the number of bytes specified when the block was allocated (or reallocated, in the case of **realloc**). After the call, the freed block is available for allocation.

A null pointer argument is ignored.

In large data models (compact- and large-model programs), **free** maps to **_ffree**. In small data models (small- and medium-model programs), **free** maps to **_nfree**.

The **_ffree** function deallocates a memory block outside the default data segment. The argument *buffer* points to a memory block previously allocated through a call to **_fmalloc**. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is again available for allocation.

The **_nfree** function deallocates a memory block. The argument *buffer* points to a memory block previously allocated through a call to **_nmalloc**. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is again available for allocation.

Return Value

There is no return value.

See Also

calloc, **_fmalloc**, **malloc**, **_nmalloc**, **realloc**

Note

> Attempting to free an invalid pointer may affect subsequent allocation and cause errors. An invalid pointer is one not allocated with the appropriate call. Only blocks allocated with **calloc**, **malloc**, or **realloc** can be freed with **free**, only blocks allocated with **_fmalloc** can be freed with **_ffree**, and only blocks allocated with **_nmalloc** can be freed with **_nfree**.

Example

```
#include <malloc.h>
#include <stdio.h>

void *alloc;

main()
 {
  /* If there is nothing to free... */
  if ((alloc = malloc(100)) == NULL)
    printf("Unable to allocate memory");
  else
   {
    /* Free memory for the heap: */
    free(alloc);
    printf("100 bytes freed\n");
   }
 }
```

This program uses **malloc** to allocate a block of memory and then uses **free** to free this block.

Summary

#include <malloc.h>	Required only for function declarations
unsigned int _freect(*size*);	
size_t *size*;	Item size in bytes

Description

The **_freect** function tells you how much memory is available for dynamic memory allocation. It does so by returning the approximate number of times your program can call **malloc** to allocate an item *size* bytes long in the default data segment.

Return Value

The **_freect** function returns the number of calls as an unsigned integer.

See Also

calloc, **_expand**, **malloc**, **_memavl**, **_msize**, **realloc**

Example

```
#include <malloc.h>

main()
 {
  int i;

  /* First report on the free space: */
  printf("Approximate # of times program can call malloc\n");
  printf("to allocate a single integer = %u\n",
         _freect(sizeof(int)));

  /* Allocate space for 1000 integers: */
  for (i = 0; i < 1000; ++i)
    malloc(sizeof(int));

  /* Report again on the free space: */
  printf("Approximate # of times program can call malloc\n");
  printf("to allocate a single integer = %u\n",
         _freect(sizeof(int)));
 }
```

Sample output:

```
Approximate # of times program can call malloc
to allocate a single integer = 15268

Approximate # of times program can call malloc
to allocate a single integer = 14266
```

This program determines how much free space is available for integers in the default data segment. Then it allocates space for 1000 integers and checks the space again using **_freect**.

Summary

#include <stdio.h>

FILE *freopen(*path, type, stream*);	
const char ****path*;**	Path name of new file
const char ****type*;**	Type of access permitted
FILE ****stream*;**	Pointer to **FILE** structure

Description

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *path*. The **freopen** function is typically used to redirect the preopened files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** to files specified by the user. The new file associated with *stream* is opened with *type*, which is a character string specifying the type of access requested for the file, as follows:

Type	Description
"r"	Opens for reading. If **r** is the first character in the type string and the file does not exist or cannot be found, the **freopen** call will fail.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending); creates the file first if it doesn't exist.
"r+"	Opens for both reading and writing. (The file must exist.)
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Opens for reading and appending; creates the file first if it doesn't exist.

Note

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** types, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the **"r+"**, **"w+"**, or **"a+"** type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching between reading and writing, there must be an intervening **fsetpos**, **fseek**, or **rewind** operation. The current position can be specified for the **fsetpos** or **fseek** operation, if desired.

In addition to the values listed above, one of the following characters may be appended to the type string or inserted before the + character to specify the translation mode for new lines. For example, **r+b** is the same as **rb+**.

Mode	Meaning
t	Open in text (translated) mode; carriage-return–line-feed combinations are translated into a single line feed on input; line-feed characters are translated to carriage-return–line-feed combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading, or writing and reading, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the **fseek** and **ftell** functions to move within a file may cause **fseek** to behave improperly near the end of the file. The **t** option is not part of the ANSI standard for **freopen**, but is a Microsoft extension that should not be used where ANSI portability is desired.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in the type string, the translation mode is defined by the default mode variable **_fmode**.

Return Value

The **freopen** function returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a null pointer value.

See Also

fclose, **fcloseall**, **fdopen**, **fileno**, **fopen**, **open**, **setmode**

Example

```
#include <stdio.h>
#include <process.h>

FILE *stream, *errstream;

main()
 {

  /* Reassign "stdout" to "data2": */
  stream = freopen("data2", "w", stdout);

  /* If reassignment failed: */
  if (stream == NULL )
     fprintf("error on freopen\n");
   else
    {
     fprintf(stream,"This will go to the file 'data2'\n");
     fprintf(stream,"'stdout' successfully reassigned\n");
     system("type data2");
    }
 }
```

This program reassigns **stdout** to the file named `data2` and writes a line to that file.

frexp

Summary

#include <math.h>

double frexp(*x*, *expptr*);	
double *x*;	Floating-point value
int **expptr*;	Pointer to stored integer exponent

Description

The **frexp** function breaks down the floating-point value (*x*) into a mantissa (*m*) and an exponent (*n*) such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0, and $x = m*2^n$. The integer exponent *n* is stored at the location pointed to by *expptr*.

Return Value

The **frexp** function returns the mantissa. If *x* is 0, the function returns 0 for both the mantissa and exponent. There is no error return.

See Also

ldexp, **modf**

Example

```
#include <math.h>

main()
 {
  double x, y;
  int n;
  x = 16.4;
  y = frexp(x,&n);                /* y is .5125 and n is 5 */
  printf("y = %f and n = %d",y,n);
 }
```

This program calculates `frexp(16.4,&n)`, then displays `y` and `n`.

Summary

#include <stdio.h>

int fscanf(*stream, format*⟦, *argument*⟧...);	
FILE **stream*;	Pointer to **FILE** structure
const char **format*;	Format-control string

Description

The **fscanf** function reads data from the current position of *stream* into the locations given by *arguments* (if any). Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format.* The format controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function; see the **scanf** reference page for a description of *format.*

Return Value

The **fscanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

See Also

cscanf, **fprintf**, **scanf**, **sscanf**

Example

```
#include <stdio.h>

FILE *stream;
long l;
float fp;
char s[81];
char c;

int result;

main()
 {
  stream = fopen("data", "w+");

  /* Write data to the file: */
  fprintf(stream,"%s %ld %f%c%c","a-string",
          65000, 3.14159,'x',EOF);

  /* Set pointer to beginning of file: */
  fseek(stream,0,SEEK_SET);

  /* Read data back from file: */
  result = fscanf(stream, "%s",s   );
  result = fscanf(stream, "%ld", &l);
  result = fscanf(stream, "%f", &fp);
  result = fscanf(stream, "%c", &c );

  /* Output data read: */
  printf("%s\n", s );
  printf("%ld\n", l);
  printf("%f\n", fp);
  printf("%c\n", c );
 }
```

This program first opens a file named `data`. It then uses **fscanf** to accept various types of input data and **printf** to display these data on the screen.

Summary

#include <stdio.h>

int fseek(*stream, offset, origin***);**	
FILE **stream***;**	Pointer to **FILE** structure
long *offset***;**	Number of bytes from *origin*
int *origin***;**	Initial position

Description

The **fseek** function moves the file pointer (if any) associated with *stream* to a new location that is *offset* bytes from *origin*. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write.

The argument *origin* must be one of the following constants defined in **stdio.h**:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

The **fseek** function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However, an attempt to position the pointer in front of the beginning of the file causes an error.

The **fseek** function clears the end-of-file indicator and negates the effect of any prior **ungetc** calls against *stream*.

Note

When a file is opened for appending data, the current file position is determined by the last I/O operation, not where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

For streams opened in text mode, **fseek** has limited use because carriage-return–line-feed translations can cause **fseek** to produce unexpected results. The only **fseek** operations guaranteed to work on streams opened in text mode are the following:

- Seeking with an offset of 0 relative to any of the origin values
- Seeking from the beginning of the file with an offset value returned from a call to **ftell**

Return Value

If successful, **fseek** returns 0. Otherwise, it returns a nonzero value. On devices incapable of seeking, the return value is undefined.

See Also

ftell, **lseek**, **rewind**

Example

```
#include <stdio.h>

FILE *stream;
main()
{
 char line[81];
 int result;
 stream = fopen("data","w+");
 fprintf(stream,"This is the first line in file 'data'.\n");
 result = fseek(stream,0L,SEEK_SET); /* Position pointer */
 if (result)
  perror("Fseek failed");
 else {
  printf("File pointer is set to the beginning of file.\n");
  fgets(line,80,stream);
  printf("%s",line);
 }
}
```

This program opens the file `data` and moves the pointer to the file's beginning.

Summary

#include <stdio.h>

int fsetpos(*stream, pos*);	
FILE *_stream_;	Target stream
const fpos_t *_pos_;	Position-indicator storage

Description

The **fsetpos** function sets the file-position indicator for *stream* to the value of *pos*, which is obtained in a prior call to **fgetpos** against *stream*.

The function clears the end-of-file indicator and undoes any effects of the **ungetc** function on *stream*. After calling **fsetpos**, the next operation on *stream* may be either input or output.

Return Value

If successful, the **fsetpos** function returns 0. On failure, the function returns a nonzero value and sets **errno** to one of the following manifest constants (defined in **stdio.h**):

Constant	Meaning
EINVAL	An invalid *stream* value was passed.
EBADF	The object that *stream* points to is not a valid file handle, or the file is not accessible.

See Also

fgetpos

Example

```
#include <stdio.h>

FILE *stream;
fpos_t *pos;
int val;
char list[100];

main()
{
 stream = fopen("file1","rb");                          /* Open file1 */
 fread(list,sizeof(char),100,stream);           /* Read some data */
 if (fgetpos(stream,pos) != 0)          /* Save current position */
  perror("fgetpos error");
 fread(list,sizeof(char),100,stream);           /* Read some more */
 if (fsetpos(stream,pos) != 0)      /* Return to saved position */
  perror("fsetpos error");
}
```

This program opens the file named `file1` and reads 100 characters. It then uses **fgetpos** to find and save the file position pointer. After performing another read, the program calls **fsetpos** to restore the file pointer to the saved position.

Summary

#include <sys\types.h>
#include <sys\stat.h>

int fstat(*handle, buffer*);	
int *handle*;	Handle of open file
struct stat **buffer*;	Pointer to structure to store results

Description

The **fstat** function obtains information about the open file associated with *handle* and stores it in the structure that *buffer* points to. The structure, whose type **stat** is defined in **sys\stat.h**, contains the following fields:

Field	Value
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).
st_dev	Either the drive number of the disk containing the file, or *handle* in the case of a device (same as **st_rdev**).
st_mode	Bit mask for file-mode information. **S_IFCHR** bit set if *handle* refers to a device. **S_IFREG** bit set if *handle* refers to an ordinary file. User read/write bits set according to the file's permission mode.
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_nlink	Always 1.
st_rdev	Either the drive number of the disk containing the file, or *handle* in the case of a device (same as **st_dev**).
st_size	Size of the file in bytes.

There are three additional fields in the **stat** structure type that do not contain meaningful values under DOS.

Return Value

The **fstat** function returns the value 0 if the file-status information is obtained. A return value of –1 indicates an error; in this case, **errno** is set to **EBADF**, indicating an invalid file handle.

See Also

access, **chmod**, **filelength**, **stat**

Note

If *handle* refers to a device, the size and time fields in the **stat** structure are not meaningful.

Example

```
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <io.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";
```

```
main()
 {
  fh = open("data", O_CREAT | O_WRONLY | O_TRUNC);
  write(fh,buffer,strlen(buffer));

  /* Get data associated with "fh": */

  result = fstat(fh,&buf);

  /* Check if statistics are valid: */
  if (result != 0)
    printf("Bad file handle\n" );

  /* Output some of  */
  /* the statistics: */
  else
   {
    printf("File size     : %ld\n",buf.st_size);
    printf("Drive number  : %d\n",buf.st_dev);
    printf("Time modified : %s",ctime(&buf.st_atime));
   }
 }
```

This program uses **fstat** to report the size of a file named `data`.

ftell

Summary

# include <stdio.h>

long ftell(*stream*);	
FILE *_stream_;	Target **FILE** structure

Description

The **ftell** function gets the current position of the file pointer (if any) associated with *stream*. The position is expressed as an offset relative to the beginning of the stream.

Note

> When a file is opened for appending data, the current file position is determined by the last I/O operation, not where the next write would occur. For example, if a file is opened for an append and the last operation was a read, the file position is the point where the next read operation would start, not where the next write would start. If no I/O operation has yet occurred on a file opened for appending, then the file position is the beginning of the file.

Return Value

The **ftell** function returns the current position. On error, the function returns –1L and **errno** is set to one of the following constants, defined in **errno.h**:

Constant	Description
EBADF	Bad file number. The *stream* argument is not a valid file-handle value or does not refer to an open file.
EINVAL	Invalid argument. An invalid *stream* argument was passed to the function.

On devices incapable of seeking (such as terminals and printers), or when *stream* does not refer to an open file, the return value is undefined.

See Also

fgetpos, **fseek**, **lseek**, **tell**

Note

> The value returned by **ftell** may not reflect the physical byte offset for streams opened in text mode, since text mode causes carriage-return–line-feed translation. Use **ftell** in conjunction with the **fseek** function to remember and return to file locations correctly.

Example

```
#include <stdio.h>

FILE *stream;
long position;
char list[100];

main()
 {
  stream = fopen("data","rb");
  /* Move the pointer by reading data: */
  fread(list,sizeof(char),100, stream);
  /* Get position after read: */
  position = ftell(stream);
  printf("position = %ld\n", position);
 }
```

This program opens a file named `data` for reading and tries to read 100 characters. It then uses **ftell** to determine the position of the file pointer and displays this position.

ftime

Summary

#include <sys\types.h>
#include <sys\timeb.h>

void ftime(*timeptr***);**	
struct timeb ****timeptr*;**	Pointer to structure defined in **sys\timeb.h**

Description

The **ftime** function gets the current time and stores it in the structure pointed to by *timeptr*. The **timeb** structure is defined in **sys\timeb.h**. It contains four fields, **time**, **millitm**, **timezone**, and **dstflag**, which have the following values:

Field	Value
dstflag	Nonzero if daylight saving time is currently in effect for the local time zone. (See **tzset** for an explanation of how daylight saving time is determined.)
millitm	Fraction of a second in milliseconds.
time	The time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.
timezone	The difference in minutes, moving westward, between Greenwich mean time and local time. The value of **timezone** is set from the value of the global variable **timezone** (see **tzset**).

Return Value

The **ftime** function gives values to the fields in the structure pointed to by *timeptr*. It does not return a value.

See Also

asctime, **ctime**, **gmtime**, **localtime**, **time**, **tzset**

Example

```
#include <sys/types.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <time.h>

main()
 {
  struct timeb timebuffer;
  char *timeline;

  ftime(&timebuffer);
  timeline = ctime(&(timebuffer.time));

  printf("The time is %.19s.%hu %s",timeline,
         timebuffer.millitm, &timeline[20]);
 }
```

Sample output:

```
The time is Wed Dec 04 17:58:29.420 1985
```

This program uses **ftime** to obtain the current time and then stores this time in `timebuffer`.

fwrite

Summary

#include <stdio.h>

size_t fwrite(*buffer, size, count, stream***);**

const void **buffer*;	Pointer to data to be written
size_t *size*;	Item size in bytes
size_t *count*;	Maximum number of items to be written
FILE **stream*;	Pointer to **FILE** structure

Description

The **fwrite** function writes up to *count* items of length *size* from *buffer* to the output *stream*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually written.

If *stream* is opened in text mode, each carriage return is replaced with a carriage-return–line-feed pair. The replacement has no effect on the return value.

Return Value

The **fwrite** function returns the number of full items actually written, which may be less than *count* if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined.

See Also

fread, **write**

Example

```
#include <stdio.h>

FILE *stream;
long list[100];
int numread;
int numwritten;

main()
 {
  /* File opened in "binary" mode: */
  if ((stream = fopen("data.bin", "w+b")) != NULL) {
    /* Write 100 long integers to "stream": */
    numwritten = fwrite((char *)list,sizeof(long),100,stream);
    printf( "Wrote %d items\n", numwritten );
   }
  else
    printf("Problem opening the file");
  fclose(stream);
  if ((stream = fopen("data.bin", "r+b")) != NULL) {
    /* Attempt to read in 100 long ints: */
    numread = fread((char *)list,sizeof(long),100,stream);
    printf("Number of items read = %d\n", numread);
   }
  else
    printf("Was not able to open the file");
 }
```

This program opens a file named `data.bin` and writes 100 long integers to the file. The program then tries to open `data.bin` and read in 100 long integers. If the attempt succeeds, the program displays the number of actual items read.

gcvt

Summary

#include <stdlib.h>	Required only for function declarations
char *gcvt(*value*, *digits*, *buffer***);**	
double *value***;**	Value to be converted
int *digits***;**	Number of significant digits stored
char ****buffer*;**	Storage location for result

Description

The **gcvt** function converts a floating-point *value* to a character string and stores the string in *buffer*. The *buffer* should be large enough to accommodate the converted value plus a terminating null character (**'\0'**), which is automatically appended. There is no provision for overflow.

The **gcvt** function attempts to produce *digits* significant digits in decimal format. Failing that, it produces *digits* significant digits in exponential format. Trailing zeros may be suppressed in the conversion.

Return Value

The **gcvt** function returns a pointer to the string of digits. There is no error return.

See Also

atof, **atoi**, **atol**, **ecvt**, **fcvt**

■ Example

```
#include <stdlib.h>
#include <stdio.h>

char buffer[50];
int precision = 7;

main()
 {

  gcvt(-3.1415e5,precision,buffer);
  /* buffer now contains "-314150." */
  printf( "buffer= \"%s\"\n", buffer );
 }
```

This program converts `-3.1415e5` to its string representation, then displays this string.

_getbkcolor

Summary

#include <graph.h>

long far _getbkcolor(void);

Description

The **_getbkcolor** function retrieves the pixel value of the current background color. The default is 0.

Return Value

The function returns the current background color value. There is no error return.

See Also

_setbkcolor

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop;
 _setvideomode(_MRES16COLOR);
 for (loop = 0; loop < 20; loop++ ) {
  /* Get the next background color */
  _setbkcolor((_getbkcolor() + 1 ) % 16 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program repeatedly sets a new background color.

Summary

#include <stdio.h>

int getc(*stream*);	Reads a character from *stream*
FILE ****stream*;**	Pointer to **FILE** structure
int getchar(void);	Reads a character from **stdin**

Description

The **getc** macro reads a single character from the current *stream* position and increases the associated file pointer (if there is one) to point to the next character. The **getchar** macro is identical to **getc(stdin)**.

Return Value

The **getc** and **getchar** macros return the character read. A return value of **EOF** indicates an error or end-of-file condition. Use **ferror** or **feof** to determine whether an error or end-of-file occurred.

See Also

fgetc, **fgetchar**, **getch**, **getche**, **putc**, **putchar**, **ungetc**

Note

The **getc** and **getchar** routines are similar to **fgetc** and **fgetchar**, respectively, but are macros, not functions.

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i, ch;

main()
 {
  stream = fopen("getc.c", "r");

  printf("Enter a line >> ");

     /* Read in single line from "stdin": */
  for (i = 0; (i < 80) && ((ch = getchar()) != EOF)
       && (ch != '\n'); i++)

    buffer[i] = ch;

  /* Terminate string with null character: */
  buffer[i] = '\0';
  printf("%s\n", buffer);
 }
```

This program uses **getchar** to read a single line of input from **stdin**, places this input in `buffer`, then terminates the string before printing it to the screen.

Summary

#include <conio.h>	Required only for function declarations
int getch(void);	Reads character without echo
int getche(void);	Reads character and echo

Description

The **getch** function reads without echoing a single character from the console. The **getche** function reads a single character from the console and echoes the character read. Neither function can be used to read CTRL+C.

When reading a function key or cursor-moving key, the **getch** and **getche** functions must be called twice; the first call returns 0 or EOH and the second call returns the actual key code.

Return Value

The **getch** function returns the character read. There is no error return.

See Also

cgets, **getchar**, **ungetch**

Example

```
#include <conio.h>
#include <ctype.h>

int ch;
main()
{
 printf("Input whitespace characters, "
        "followed by a non-whitespace character\n");
 do ch = getch();
 while (isspace(ch));
 putch(ch);
}
```

This program reads characters from the keyboard, but does not echo them until it reads the first nonblank character.

_getcolor

Summary

#include <graph.h>

short far _getcolor(void);

Description

The **_getcolor** function returns the pixel value of the current color. The default is the highest legal value of the current palette.

Return Value

There is no error return.

See Also

_setcolor

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop, loop1;
 _setvideomode(_MRES16COLOR);
 for (loop1 = 0; loop1 < 20; loop1++) { /* Get next color: */
     _setcolor((_getcolor() + 1) % 16 );
     for (loop = 0; loop < 3200; loop++) {
         /* set a random pixel normalized to be on screen */
         _setpixel( rand(1) / 104, rand(1) / 164 );
     }
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program assigns different colors to randomly selected pixels.

Summary

#include <graph.h>

struct xycoord {
    short xcoord; x coordinate
    short ycoord; y coordinate
    } far _getcurrentposition(void);

Description

The **_getcurrentposition** function returns the logical coordinates of the current graphics output position as an **xycoord** structure, defined in **graph.h**.

The current position can be changed by the **_arc** and **_moveto** functions.

Only graphics output starts at the current position; it does not affect text output, which begins at the current text position, a separate concept (see **_settextposition**).

Return Value

There is no error return.

See Also

_moveto

■ Example

```
#include <stdio.h>
#include <graph.h>

struct xycoord position;
char buffer[ 255 ];

main()
{
 int loop;
 _setvideomode(_MRES16COLOR);
 _moveto( rand(1) % 160 + 80, rand(1) % 100 + 50 );
 position = _getcurrentposition();
 sprintf(buffer,"x=%d, y=%d",position.xcoord,position.ycoord );
 _settextposition( position.xcoord / 8, position.ycoord / 8 );
 _outtext( buffer );
 while (!kbhit()); /* wait for key to reset */
 _setvideomode (_DEFAULTMODE);
}
```

This program moves the current graphics output position to a random point, calls **_getcurrentposition** to obtain the coordinates, and then writes the coordinates to a buffer. It then sets the current text position to those coordinates and outputs the coordinates to the screen, beginning at the current position.

Summary

#include <direct.h>	Required only for function declarations
char *getcwd(*path, n***);**	
char ****path*;**	Storage location for path name
int *n***;**	Maximum length of path name

Description

The **getcwd** function gets the full path name of the current working directory and stores it at *path.* The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name (including the terminating null character) exceeds *n.*

The *path* argument can be **NULL**; a buffer of at least size *n* (more only if necessary) will automatically be allocated using **malloc** to store the path name. This buffer can later be freed by calling **free** and passing it the **getcwd** return value (a pointer to the allocated buffer).

Return Value

The **getcwd** function returns *path.* A **NULL** return value indicates an error, and **errno** is set to one of the following values:

Value	Meaning
ENOMEM	Insufficient memory to allocate *n* bytes (when a **NULL** argument is given as *path*)
ERANGE	Path name longer than *n* characters

See Also

chdir, **mkdir**, **rmdir**

Example

```
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

main()
 {
  char buffer[67];

  /* Get the current working directory: */
  if (getcwd(buffer,66) == NULL)
    perror("getcwd error");
  else
     printf("%s",buffer);
 }
```

This program places the name of the current directory in the `buffer` array, then displays the name of the current directory on the screen. Specifying a length of 66 characters for `buffer` allows room for the longest legal directory name plus two characters for the drive specification.

Summary

#include <stdlib.h>	Required only for function declarations
char *getenv(*varname*);	
const char **varname*;	Name of environment variable

Description

The **getenv** function searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes. (For example, the **LIB** environment variable defines the default search path for libraries to be linked with a program.)

Return Value

The **getenv** function returns a pointer to the environment table entry containing the current string value of *varname*. The return value is **NULL** if the given variable is not currently defined.

See Also

putenv

Note

Environment-table entries must not be changed directly. If an entry must be changed, use the **putenv** function. To modify the returned value without affecting the environment table, use **strdup** or **strcpy** to make a copy of the string.

The **getenv** and **putenv** functions use the global variable **environ** to access the environment table. The **putenv** function may change the value of **environ**, thus invalidating the *envp* argument to the **main** function. Therefore, it's safer to use the **environ** variable to access the environment table.

Example

```
#include <stdlib.h>
#include <stdio.h>

char *pathvar;

main()
 {
  /* Get the value of the PATH environment variable:     */
  pathvar = getenv("PATH");
  printf("%s\n", pathvar ? pathvar : "path variable not set");
 }
```

This program uses **getenv** to retrieve the **PATH** environment variable and then displays its value. If PATH=`A:\BIN;B:\BIN` is in the environment, `pathvar` points to `A:\BIN;B:\BIN`. If there is no **PATH** environment variable, `pathvar` is **NULL**.

_getfillmask

Summary

#include <graph.h>

unsigned char far * far _getfillmask(*mask*);
**unsigned char far ****mask*; Mask array

Description

Some graphics routines (**_floodfill**, **_pie**, **_ellipse**, and **_rectangle**) can fill part or all of the screen with the current color or background color. The filling can be controlled with the current fill mask.

The **_getfillmask** function returns the current fill mask. The mask is an 8-by-8 array of bits, where each bit represents a pixel. A 1 bit sets the corresponding pixel to the current color, while a 0 bit leaves the pixel unchanged. The mask is repeated over the entire fill area. If no fill mask is set, or *mask* is **NULL**, only the current color is used in fill operations.

Return Value

If no mask is present, the function returns **NULL**.

See Also

_floodfill, **_setfillmask**.

Example

```
#include <stdio.h>
#include <graph.h>

unsigned char *(style[ 6 ]) = { "x00x00x00x00x00x00x00x00",
    "x20x08x20x08x20x08x20x08", "x98xc6x30x30x8cx4cx62x18",
    "xe6x38xb2x9cxe6x38xb2x9c", "xfcxeex7axdexf6xbcxeex7a",
    "xfexfexfexfexfexfexfexfe" };

char *oldstyle = "12345678";   /* place holder for old style */
```

```
main()
{
 int loop;
 _setvideomode(_MRES4COLOR);
 _getfillmask( oldstyle );
 _setcolor( 2 );
 /* draw an ellipse under the middle few rectangles */
 /* in a different color */
 _ellipse( _GFILLINTERIOR, 120, 75, 200, 125 );
 _setcolor( 3 );
 for ( loop = 0; loop < 6; loop++ ) {
  /* make 6 rectangles, the first background color */
  _setfillmask( (char far *)(style[ loop ]) );
  _rectangle(_GFILLINTERIOR,loop*40+5,90,(loop+1)*40,110 );
 }
 _setfillmask( oldstyle );    /* restore old style */
 while( !kbhit() ); /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws an ellipse overlaid with six rectangles, each with a different fill mask.

Summary

#include <graph.h>

void far _getimage(*x1, y1, x2, y2, image*);

short *x1, y1*;	Upper-left corner of bounding rectangle
short *x2, y2*;	Lower-right corner of bounding rectangle
char far **image*;	Storage buffer for screen image

Description

The **_getimage** function stores the screen image in the bounding rectangle defined by the logical points (*x1, y1*) and (*x2, y2*) into the buffer that *image* points to. The buffer must be large enough to hold the image. You can determine the size by calling **_imagesize** at run time, or by using the formula described in the **_imagesize** reference page.

Return Value

No value is returned.

See Also

_imagesize, **_putimage**

Example

```
#include <stdio.h>
#include <malloc.h>
#include <graph.h>

char far *buffer;
```

```
main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 for ( xvar = 163, loop = 0; xvar < 320; loop++, xvar += 3 ) {
  _setcolor(loop % 16 );
  yvar = xvar * 5 / 8;
  _rectangle(_GBORDER, 320-xvar, 200-yvar, xvar, yvar);
  _setcolor(rand(1) % 16 );
  _floodfill(0, 0, loop % 16 );
 }
 buffer = (char far *)malloc( (unsigned int)
          _imagesize( 0, 0, 80, 50 ) );
 if ( buffer == (char far *)NULL ) {
  exit( -1 );
 }
 _getimage(0, 0, 80, 50, buffer );
 _putimage( 80, 50, buffer, _GXOR );
 free((char *)buffer);
 while ( !kbhit() ); /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a rectangle and then calls **_getimage** to store it in memory.

Summary

#include <graph.h>

unsigned short far _getlinestyle(void);

Description

Some graphics routines (**_lineto** and **_rectangle**) output straight lines to the screen. The type of line can be controlled with the current line-style mask.

The **_getlinestyle** function returns the current line-style-mask number. The mask is a 16-bit array, where each bit represents a pixel in the line being drawn. If the bit is 1, the corresponding pixel is set to the color of the line (the current color). If the bit is 0, the corresponding pixel is left unchanged. The mask is repeated over the length of the line. The default mask is 0xFFFF (a solid line).

Return Value

If no mask has been set, **_getlinestyle** returns the default mask.

See Also

_lineto, **_pie**, **_rectangle**, **_setlinestyle**

Example

```
#include <stdio.h>
#include <graph.h>

short style[16] = {0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff,
    0x1ff, 0x3ff, 0x7ff, 0xfff, 0x1fff, 0x3fff, 0x7fff,
    0xffff};
```

```
main()
{
 int xvar, yvar, loop, oldstyle;
 _setvideomode(_MRES16COLOR);
 oldstyle = _getlinestyle(); /* save the old style of line */
 for (xvar = 0, loop = 0; xvar < 320; xvar += 3, loop++) {
  _setcolor( loop % 16 );
  yvar = xvar * 5 / 8;
  _setlinestyle( style[ loop % 16 ] );
  _rectangle( _GBORDER, 320 - xvar, 200 - yvar, xvar, yvar );
 }
 _setlinestyle(oldstyle);
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_getlinestyle** to preserve the current line style before changing it for the subsequent rectangle output.

Summary

#include <graph.h>

struct xycoord {
  short xcoord; x coordinate
  short ycoord; y coordinate
  } **far _getlogcoord(***x*, *y***);**
short *x*, *y***;** Physical point to translate

Description

The **_getlogcoord** function translates the physical coordinates (*x*, *y*) to logical coordinates and returns them in an **xycoord** structure, defined in **graph.h**.

Return Value

There is no error return.

See Also

_getphyscoord, **_moveto**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 struct xycoord xycoord;
 int loop;
 _setvideomode(_MRES16COLOR);
 xycoord.xcoord = rand(1) % 320;
 xycoord.ycoord = rand(1) % 200;
 xycoord = _getphyscoord(xycoord.xcoord, xycoord.ycoord);
 /* set the logical origin to a random place on the screen */
 _setlogorg(xycoord.xcoord, xycoord.ycoord);
 /* draw an ellipse around this random origin */
 _ellipse(_GBORDER, -80, -50, 80, 50);
 xycoord = _getlogcoord(0, 0);
 _moveto(xycoord.xcoord, xycoord.ycoord);
 xycoord = _getlogcoord( 320, 200 );
 _lineto(xycoord.xcoord, xycoord.ycoord);
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_getphyscoord** to find the physical coordinates of a randomly selected logical point, to which it then redefines the logical origin. The program draws an ellipse around the logical origin. It then calls **_getlogcoord** to find the logical coordinates of the physical origin, moves the current output position to the physical origin, calls **_getlogcoord** again to find the logical coordinates of another point, and finally draws a straight line from the physical origin to that point.

Summary

```
#include <graph.h>

struct xycoord {
      short xcoord;   x coordinate
      short ycoord;   y coordinate
      } far _getphyscoord(x, y);
short x, y;        Logical point to translate
```

Description

The **_getphyscoord** function translates the logical point (*x*, *y*) to physical coordinates, returning them in an **xycoord** structure, defined in **graph.h**.

Return Value

There is no error return.

See Also

_getlogcoord

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 struct xycoord xycoord;
 int loop;
 _setvideomode(_MRES16COLOR);
 xycoord.xcoord = rand(1) % 320;
 xycoord.ycoord = rand(1) % 200;
 xycoord = _getphyscoord(xycoord.xcoord, xycoord.ycoord);
 /* Set the logical origin to a random place on the screen: */
 _setlogorg(xycoord.xcoord, xycoord.ycoord);
 /* Draw an ellipse around this random origin: */
 _ellipse(_GBORDER, -80, -50, 80, 50);
 xycoord = _getlogcoord(0, 0);
 _moveto(xycoord.xcoord, xycoord.ycoord);
 xycoord = _getlogcoord( 320, 200 );
 _lineto(xycoord.xcoord, xycoord.ycoord);
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_getphyscoord** to find the physical coordinates of a randomly selected logical point, to which it then redefines the logical origin. The program draws an ellipse around the logical origin. It then calls **_getlogcoord** to find the logical coordinates of the physical origin, moves the current output position to the physical origin, calls **_getlogcoord** again to find the logical coordinates of another point, and finally draws a straight line from the physical origin to that point.

Summary

#include <process.h> Required only for function declarations

int getpid(void);

Description

The **getpid** function returns an integer (the process ID) that uniquely identifies the calling process.

Return Value

The **getpid** function returns the process ID. There is no error return.

See Also

mktemp

Example

```
#include <process.h>
#include <string.h>
#include <stdio.h>

char filename[9], pid[5];

main()
 {
  strcpy(filename, "FILE");
  strcat(filename, itoa(getpid(), pid,10));

  /* Prints "FILExxxxx", where xxxxx */
  /* is the process ID:             */
  printf("Filename is %s\n", filename);
 }
```

This program uses **getpid** to obtain the process ID, then converts the process ID to a string for output.

_getpixel

Summary

#include <graph.h>

short far _getpixel(*x*, *y*);
short *x*, *y*;   Pixel position

Description

The **_getpixel** function retrieves the pixel value at the logical point (*x*, *y*). The range of possible pixel values and their color translation is determined by the current video mode and palette, respectively.

Return Value

If successful, the function returns the pixel value. If the function fails (for example, the point lies outside of the clipping region), it returns –1.

See Also

_remapallpalette, **_remappalette**, **_selectpalette**, **_setpixel**, **_setvideomode**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 _rectangle(_GFILLINTERIOR, 80, 50, 240, 150 );
 for (loop = 0; loop < 8000L; loop++) {
  /* Fill pixels at random, but only if they are already on */
  if (_getpixel(xvar = rand(1) / 104, yvar = rand(1)/164)) {
   _setcolor(rand(1) % 16);
   _setpixel(xvar, yvar);
  }
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program assigns different colors to randomly selected pixels.

Summary

#include <stdio.h>

char *gets(*buffer*);
char **buffer*; Storage location for input string

Description

The **gets** function reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first new-line character (**'\n'**). The **gets** function then replaces the new-line character with a null character (**'\0'**) before returning the line. In contrast, the **fgets** function retains the new-line character.

Return Value

If successful, the **gets** function returns its argument. A null pointer indicates an error or end-of-file condition. Use **ferror** or **feof** to determine which one has occurred.

See Also

fgets, **fputs**, **puts**

Example

```
#include <stdio.h>

char line[100];
char *result;

main()
 {
  printf("Input a string: ");
  result = gets(line);
  printf("The line entered was: %s\n", result);
 }
```

This program uses **gets** to read a line of input from **stdin**.

_gettextcolor

Summary

#include <graph.h>

short far _gettextcolor(void);

Description

The **_gettextcolor** function returns the pixel value of the current text color. The default is the highest legal value of the current palette.

Return Value

There is no error return.

See Also

_selectpalette, **_settextcolor**

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];
```

```
main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);     /* turn wrapping off */
 oldcolor = _gettextcolor(); /* save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* recover original color */
 _outtext("This should be on last line; is out of the window");
 while (!kbhit());  /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_gettextcolor** to save the current text color before manipulating the screen.

_gettextposition

Summary

```
#include <graph.h>

struct rccoord {
      short row;      Row coordinate
      short col;      Column coordinate
      } far _gettextposition(void);
```

Description

The **_gettextposition** function returns the current text position as an **rccoord** structure, defined in **graph.h**.

Text output begins at the current text position. Graphics output begins at the current graphics output position, which is a separate position.

Return Value

There is no error return.

See Also

_settextposition

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];

main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen: */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);      /* turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on last line; is out of the window");
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_gettextposition** and assigns the return value to the structure `rcoord`. It increments the row position and prints the new coordinates.

_getvideoconfig

Summary

```
#include <graph.h>

struct videoconfig {
      short  numxpixels;          Number of pixels in x axis
      short  numypixels;          Number of pixels in y axis
      short  numtextcols;         Number of text columns available
      short  numtextrows;         Number of text rows available
      short  numcolors;           Number of actual colors
      short  bitsperpixel;        Number of bits representing a pixel
      short  numvideopages;              Number of available video pages
      } far * far _getvideoconfig(config);
struct videoconfig far *config;   Configuration information
```

Description

The **_getvideoconfig** function returns the current graphics environment configuration in a **videoconfig** structure, defined in **graph.h**.

Return Value

There is no error return.

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 struct videoconfig config;
 _setvideomode(_MRES16COLOR);
 _getvideoconfig( &config );
 /* Set logical origin to the center of the screen: */
 _setlogorg(config.numxpixels/2-1, config.numypixels/2 - 1);
 _moveto( -80, -50 );
 _lineto( 80, 50 );
 _lineto( 80, -50 );
 while (!kbhit()); /* wait for key before restoring screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_getvideoconfig** to determine the screen size (in pixels) of the current hardware configuration. It then sets the logical origin to the center of the screen.

Summary

#include <stdio.h>

int getw(*stream*);	
**FILE ****stream*;	Pointer to **FILE** structure

Description

The **getw** function reads the next binary value of type **int** from *stream* and increases the associated file pointer (if there is one) to point to the next unread character. The **getw** function does not assume any special alignment of items in the stream.

Return Value

The **getw** function returns the integer value read. A return value of **EOF** may indicate an error or end-of-file; however, the **EOF** value is also a legitimate integer value, so **feof** or **ferror** should be used to verify an end-of-file or error condition.

See Also

putw

Note

> The **getw** function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with **getw** since the size of an **int** and the ordering of bytes within an **int** differ across systems.

Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int i;

main()
 {
  stream = fopen("data.bin", "rb");

  /* Read a word from the stream: */
  i = getw(stream);

 /* If there is an error... */
 if (ferror(stream))
  {
   printf("getw failed\n");
   clearerr(stream);
  }
 else
   printf("Word = %x\n", i);
 }
```

This program uses **getw** to read a word from a stream, then performs an error check.

Summary

#include <time.h>

struct tm *gmtime(*time***);**
const time_t ****time*;** Pointer to stored time

Description

The **gmtime** function converts the *time* value to a structure. The *time* argument represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time; this value is usually obtained from a call to **time**.

The **gmtime** function breaks down the *time* value and stores it in a structure of type **tm**, defined in **time.h**. The structure members are described in the reference page for **asctime**. The structure result reflects Greenwich mean time, not local time.

The fields of the structure type **tm** store the following values:

Field	Value Stored
tm_sec	Seconds
tm_min	Minutes
tm_hour	Hours (0–24)
tm_mday	Day of month (1–31)
tm_mon	Month (0–11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0–6; Sunday = 0)
tm_yday	Day of year (0–365; January 1 = 0)
tm_isdst	Always 0 for **gmtime**

MS-DOS does not understand dates prior to 1980. If *time* represents a date prior to 1980, **gmtime** returns **NULL**.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, if *time* represents a date before January 1, 1980, **gmtime** returns the structure representation of 00:00:00, January 1, 1980.

Return Value

The **gmtime** function returns a pointer to the structure result. There is no error return.

See Also

asctime, **ctime**, **ftime**, **localtime**, **time**

Note

The **gmtime** and **localtime** functions use a single statically allocated structure to hold the result. Each call to one of these routines destroys the result of the previous call.

Example

```
#include <time.h>
#include <stdio.h>
struct tm *newtime;
long ltime;

main()
 {
  time(&ltime);
  /* Obtain Greenwich mean time: */
  newtime = gmtime(&ltime);
  printf("Greenwich mean time is %s\n",asctime(newtime));
 }
```

This program uses **gmtime** to convert a long-integer representation of Greenwich mean time to a structure named `newtime`, then uses **asctime** to convert this structure to an output string.

Summary

#include <malloc.h>	Required only for function declarations
void huge *halloc(*n*, *size*);	
long *n*;	Number of elements
size_t *size*;	Length in bytes of each element

Description

The **halloc** function allocates a huge array from MS-DOS consisting of *n* elements, each of which is *size* bytes long. Each element is initialized to 0. If the size of the array is greater than 128K (131,072 bytes), then the size of an array element must be a power of 2.

Return Value

The **halloc** function returns a **void huge** pointer to the allocated space, guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void huge**, use a type cast on the return value. The return value is **NULL** if the request cannot be satisfied.

See Also

calloc, **free**, **hfree**, **malloc**, **realloc**

Example

```
#include <stdio.h>
#include <malloc.h>

main()
 {
  long huge *lalloc;
  lalloc = (long huge *)halloc(30000L,sizeof(long));
  if (lalloc == NULL)
    printf("Insufficient memory available");
  else
    printf("Memory successfully allocated");
 }
```

This program uses **halloc** to allocate space for 30,000 long integers.

Summary

#include <dos.h>

void _harderr(void(*fptr*)());
**far ****fptr*; New INT 0x24 handler

void _hardresume(*result***);**
int *result*; Handler return parameter

void _hardretn(*error***);**
int *error*; Error to return from

Description

The **_harderr** routine establishes the user-defined routine that *fptr* points to as the handler for INT 0x24, which is invoked when a hardware error occurs during the execution of an I/O request (for example, attempting to read from a floppy disk when the drive door isn't closed). See the *MS-DOS Programmer's Reference* for more information on INT 0x24.

The **harderr** function doesn't directly install the handler that *fptr* points to; instead, **harderr** installs a handler that calls the function that *fptr* references. The handler calls the function with the following parameters:

handler(unsigned *deverror***, unsigned** *errcode***, unsigned far ****devhdr***);**

The *deverror* argument is the device error code and contains the **AX** register value that MS-DOS passes to the INT 0x24 handler. The *errcode* argument is the **DI** register value that MS-DOS passes to the handler. The low-order byte of *errcode* can be one of the following values:

Code	Meaning
0	Attempt to write to a write-protected disk
1	Unkown unit
2	Drive not ready
3	Unknown command
4	Cyclic-redundancy-check (CRC) error in data
5	Bad drive-request structure length

6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure

The *devhdr* argument is a far pointer to a device header that contains descriptive information about the device on which the error occurred. The user-defined handler must not change the information in the device-header control block.

If the error occurred on a disk device, the high-order bit (bit 15) of the *deverror* argument will be set to 0 and the *deverror* argument will indicate the following:

Bits	Meaning	
15	Disk error if false (0).	
14	Not used	
13	"Ignore" response not allowed if false.	
12	"Retry" response not allowed if false.	
11	"Fail" response not allowed if false (MS-DOS changes "fail" to "abort").	
9-10	**Code**	**Location**
	00	MS-DOS
	01	File Allocation Table (FAT)
	10	Directory
	11	Data area
8	Read error if false; write error if true	

The low-order byte of *deverror* indicates the drive where the error occurred (0 = drive A, 1 = drive B, etc.)

If the error occurs on a device other than a disk drive, the high-order bit (bit 15) of *deverror* will be 1. The attribute word located at offset 04 in the device-header block will indicate the type of device which had the error. If bit 15 of the attribute word is 0, the error is a bad memory image of the File Allocation Table. If the bit is instead 1, the error occurred on a character device and bits 0–3 of the attribute word indicate the type of device:

Bit	Meaning
3	Current clock device
2	Current null device
1	Current standard output
0	Current standard input

The user-defined handler function can issue system calls 0x01 through 0x0C only, or 0x59. Thus, many of the standard C run-time functions (such as stream I/O and low-level I/O) cannot be used in a hardware error handler. Function 0x59 may be used to obtain further information about the error that occurred.

If the handler returns, it can do so using any of these three methods:

1. By way of the **return** statement
2. By way of the **_hardresume** function
3. By way of the **_hardretn** function

If the handler returns by way of **_hardresume** or a **return** statement, the handler returns to MS-DOS. If the handler returns by way of **_hardretn**, the handler bypasses MS-DOS and returns to the application at the point just past the failing I/O function request.

The **_hardresume** function should only be called from within the user-defined hardware error handler function. This function allows the user to return from the handler to MS-DOS, as will returning from the handler using a **return** statement.

The result supplied to **_hardresume** must be one of the following constants:

Constant	Action
_HARDERR_IGNORE	Ignore the error
_HARDERR_RETRY	Retry the operation
_HARDERR_ABORT	Abort the program issuing INT 0x23
_HARDERR_FAIL	Fail the system call that is in progress (this is not supported on MS-DOS 2.*x*)

The **_hardretn** function allows the user-defined hardware error handler to return directly to the application program rather than returning to MS-DOS. The application resumes at the point just after the failing I/O function request. The **_hardretn** function should only be called from within a user-defined hardware error handler function.

The error parameter of **_hardretn** should be an MS-DOS error code, as opposed to the XENIX-style error code that is available in **errno**. For information about the MS-DOS error codes which may be returned by a given MS-DOS function call, refer to the *MS-DOS Programmer's Reference.*

If the failing I/O function request is an INT 0x21 function greater than or equal to function 0x38, then **_hardretn** will return to the application with the carry flag set and the **AX** register set to the **_hardretn** *error* parameter. If the failing INT 0x21 function request is less than function 0x38 and the function can return an error, the **AL** register will be set to 0xFF on return to the application. If the failing INT 0x21 does not have a way of returning an error condition (this is true of certain INT 0x21 functions below 0x38), the error parameter of **_hardretn** is not used and no error code is returned to the application.

■ See Also

_chain_intr, **_dos_getvect**, **_dos_setvect**

_heapchk, _fheapchk, _nheapchk

Summary

#include <malloc.h>

int _heapchk(void);	Runs consistency check on heap
int _fheapchk(void);	Runs consistency check on far heap
int _nheapchk(void);	Runs consistency check on near heap

Description

Along with the **_heapset** and **_heapwalk** routines, **_heapchk** is provided as an aid in debugging heap-related problems in programs.

The **_heapchk** routine does a minimal consistency check on the unallocated memory space, or "heap." The consistency check determines whether all the heap entries are within the bounds of the heap's current memory allocation.

In large data models (compact- and large-model programs), **_heapchk** maps to **_fheapchk**. In small data models (small- and medium-model programs), **_heapchk** maps to **_nheapchk**. The **_fheapchk** routine runs the consistency check on the far heap, while **_nheapchk** runs the consistency check on the near heap.

Return Value

All three routines return one of the following manifest constants (defined in **malloc.h**):

Constant	Meaning
_HEAPOK	The heap appears to be consistent.
_HEAPEMPTY	The heap has not been initialized.
_HEAPBADBEGIN	The initial header information could not be found.
_HEAPBADNODE	A bad node has been found, or the heap is damaged.

See Also

_heapset, _heapwalk

Example

```
#include <malloc.h>
#include <stdio.h>

main()
{
 int heapstatus();
 char *p = malloc(100);
 malloc(200);
 malloc(300);
 free(p);

 heapstatus = _heapchk();
 switch(heapstatus){
  case _HEAPOK:
        printf("OK - heap is fine\n");
        break;
  case _HEAPEMPTY:
        printf("OK - heap is empty\n");
        break;
  case _HEAPBADBEGIN:
        printf("ERROR - bad start of heap\n");
        breal;
  case _HEAPBADNODE:
        printf("ERROR - bad node in heap\n");
        break;
 }
}
```

This program checks the heap for consistency and prints an appropriate message.

_heapset, _fheapset, _nheapset

Summary

#include <malloc.h>

int _heapset(*fill*);	Fills empty heap nodes
unsigned int *fill*;	Fill character
int _fheapset(*fill*);	Fills empty far heap nodes
unsigned int *fill*;	Fill character
int _nheapset(*fill*);	Fills empty near heap nodes
unsigned int *fill*;	Fill character

Description

Along with the **_heapchk** and **_heapwalk** routines, **_heapset** is provided as an aid in debugging heap-related problems in programs.

The **_heapset** routine first does a minimal-consistency check on the heap (just as **_heapchk** does) and then sets the heap's free entries with the *fill* value. You can use this in debugging to see where the free nodes are located in memory dumps of the heap, and also to show where data was unintentionally written to memory that was freed.

In large data models (compact and large-model programs), **_heapset** maps to **_fheapset**. In small data models (small and medium-model programs), **_heapset** maps to **_nheapset**. The **_fheapset** routine operates on the far heap, while **_nheapset** operates on the near heap.

Return Values

All three routines return an **int** whose value is one of the following manifest constants (defined in **malloc.h**):

Constant	Meaning
_HEAPOK	Heap appears to be consistent
_HEAPEMPTY	Heap has not been initialized
_HEAPBADBEGIN	The initial header information could not be found or was invalid
_HEAPBADNODE	A bad node was found, or the heap is damaged

See Also

_heapchk, _heapwalk

Example

```
#include <malloc.h>
#include <stdio.h>

main()
{
 int heapstatus;
 char *p = malloc(1);   /* make sure heap is initialized */

 heapstatus = _heapset('Z');    /* fill in free entries */
 switch(heapstatus){
  case _HEAPOK:
        printf("OK - heap is fine\n");
        break;
  case _HEAPEMPTY:
        printf("OK - heap is empty\n");
        break;
  case _HEAPBADBEGIN:
        printf("ERROR - bad start of heap\n");
        break;
  case _HEAPBADNODE:
        printf("ERROR - bad node in heap\n");
        break;
 }
}
```

This program checks the heap and fills in free entries with the character `'Z'`.

_heapwalk, _fheapwalk, _nheapwalk

Summary

#include <malloc.h>

int _heapwalk(*entry***);**	Get heap entry information
struct _heapinfo{	Structure to contain information about the next heap entry
int far *_pentry;	Heap entry pointer
size_t _size;	Size of heap entry
int _useflag;	Entry "in-use" flag
} ***entry*;**	
int _fheapwalk(*farentry***);**	Get far heap entry information
struct _heapinfo ***farentry*;**	Structure to contain information about the next far heap entry
int _nheapwalk(*nearentry***);**	Get near heap entry information
struct _heapinfo ***nearentry*;**	Structure to contain information about the next near heap entry

Description

Like the **_heapchk** and **_heapset** routines, **_heapwalk** is provided as an aid in debugging heap-related problems in programs.

The **_heapwalk** routine walks through the heap, one entry per call, returning a pointer to a **_heapinfo** structure that contains information about the next heap entry. The structure is defined in **malloc.h**.

Calls to **_heapwalk**, which return **_HEAPOK**, will set the `_useflag` field to either **_FREEENTRY** or **_USEDENTRY** (both are constants defined in **malloc.h**, as is the structure itself). To obtain this information about the first entry in the heap, pass **_heapwalk** a pointer to a **_heapinfo** structure whose **_pentry** field is **NULL**.

In large data models (compact- and large-model programs), **_heapwalk** maps to **_fheapwalk**. In small data models (small- and medium-model programs), **_heapwalk** maps to **_nheapwalk**. The **_fheapwalk** routine walks through the far heap entries, while the **_nheapwalk** routine walks through the near heap entries.

Return Value

All three routines return one of the following manifest constants (defined in **malloc.h**):

Constant	Meaning
_HEAPOK	The heap is OK so far, and the **_heapinfo** structure contains information about the next entry.
_HEAPEMPTY	The heap has not been initialized.
_HEAPBADPTR	The **_pentry** field of the entry structure does not contain a valid pointer into the heap.
_HEAPBADBEGIN	The initial header information was not found or it was invalid.
_HEAPBADNODE	A bad node was found or the heap is damaged.
_HEAPEND	The end of the heap was reached successfully.

See Also

_heapchk, **_heapset**

Example

```
#include <stdio.h>
#include <malloc.h>

main()
{
 char *p;
 heapdump(); p = malloc(59);
 heapdump();  free(p);
 heapdump(); p = malloc(330);
 heapdump();
}
```

```
heapdump()
{
 struct _heapinfo hinfo;
 int  heapstatus;

 hinfo._pentry = NULL;
 while((heapstatus = _heapwalk(&hinfo)) == _HEAPOK){
  printf("%6s block at %p of size %4.4X\n",
         (hinfo._useflag == _USEDENTRY ? "USED" : "FREE"),
        hinfo._pentry, hinfo._size);
 }
 switch(heapstatus){
  case _HEAPEMPTY:
        printf("OK - empty heap\n\n");
        break;
  case _HEAPEND:
        printf("OK - end of heap\n\n");
        break;
  case _HEAPBADPTR:
        printf("ERROR - bad pointer to heap\n\n");
        break;
  case _HEAPBADBEGIN:
        printf("ERROR - bad start of heap\n\n");
        break;
  case _HEAPBADNODE:
        printf("ERROR - bad node in heap\n\n");
        break;
 }
}
```

Sample Output:

```
OK - empty heap

  USED block at 1D71:1174 of size 003C
  FREE block at 1D71:11B2 of size 0E4C
OK - end of heap

  FREE block at 1D71:1174 of size 003C
  FREE block at 1D71:11B2 of size 0E4C
OK - end of heap

  FREE block at 1D71:1174 of size 003C
  USED block at 1D71:11B2 of size 014A
  FREE block at 1D71:12FE of size 0D00
OK - end of heap
```

This program "walks" the heap, starting at the beginning (`_pentry == NULL`). It prints out each heap entry's use, location, and size, and also prints out information about the overall state of the heap as soon as **_heapwalk** returns a value other than **_HEAPOK**.

Summary

#include <malloc.h> Required only for function declarations

void hfree(*buffer*);
void huge *buffer; Pointer to allocated memory block

Description

The **hfree** function deallocates a memory block; the freed memory is returned to MS-DOS. The *buffer* argument points to a memory block previously allocated through a call to **halloc**. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is available for allocation.

Return Value

There is no return value.

See Also

halloc

Note

> Attempting to free an invalid *buffer* (one not allocated with **halloc**) may affect subsequent allocation and cause errors.

Example

```
#include <malloc.h>
#include <stdio.h>

main()
 {
  void huge *alloc;

  alloc = halloc(80000L,sizeof(char));

  /* Test for valid pointer: */
  if (alloc != NULL){   /* Free memory for the heap: */
   hfree(alloc);
   printf("Memory successfully allocated and deallocated");
  }
  else
   printf("Insufficient memory available");
 }
```

This program allocates space for 80,000 characters, initializes this space to zeros, then uses **hfree** to deallocate the memory.

Summary

#include <math.h>

double hypot(*x*,*y*);
double *x*, *y*; Floating-point values

Description

The **hypot** function calculates the length of the hypotenuse of a right triangle, given the length of the two sides *x* and *y*. A call to **hypot** is equivalent to the following:

sqrt(*x*x* + *y***y*);**

Return Value

The function returns the length of the hypotenuse. If an overflow results, **hypot** returns **HUGE_VAL** and sets **errno** to **ERANGE**.

See Also

cabs

Example

```
#include <math.h>
#include <stdio.h>

main()
{
 double x = 3.0;
 double y = 4.0;
 printf("Hypotenuse = %2.1f\n", hypot(x,y));
}
```

This program prints the hypotenuse of a right triangle with sides of 3.0 and 4.0.

Summary

#include <graph.h>

long far _imagesize(*x1*, *y1*, *x2*, *y2***);**

short *x1*, *y1*;	Upper-left corner of bounding rectangle
short *x2*, *y2*;	Lower-right corner of bounding rectangle

Description

The **_imagesize** function returns the number of bytes needed to store the image defined by the bounding rectangle, specified by the coordinates (*x1*, *y1*) and (*x2*, *y2*). This size is determined by the following formula:

```
xwid = abs(x1-x2)+1;
ywid = abs(y1-y2)+1;
size = 4+((long)((xwid*bits-per-pixel+7)/8)*(long)ywid);
```

The bits-per-pixel value is returned from a call to **_getvideoconfig** as the **bitsperpixel** field.

Return Value

The function returns the image's storage size in bytes. There is no error return.

See Also

_getvideoconfig

Example

```
#include <stdio.h>
#include <malloc.h>
#include <graph.h>

char far *buffer;

main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 for ( xvar = 163, loop = 0; xvar < 320; loop++, xvar += 3 ) {
  _setcolor(loop % 16 );
  yvar = xvar * 5 / 8;
  _rectangle(_GBORDER, 320-xvar, 200-yvar, xvar, yvar);
  _setcolor(rand(1) % 16 );
  _floodfill(0, 0, loop % 16 );
 }
 buffer = (char far *)malloc( (unsigned int)
          _imagesize( 0, 0, 80, 50 ) );
 if ( buffer == (char far *)NULL ) {
  exit( -1 );
 }
 _getimage(0, 0, 80, 50, buffer );
 _putimage( 80, 50, buffer, _GXOR );
 free((char *)buffer);
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a series of nested rectangles. It calls **_imagesize** to determine how large a buffer it should allocate to store a portion of the nested-rectangle drawing.

inp, inpw

Summary

#include <conio.h>	Required only for function declarations
int inp(*port*);	Reads a byte
unsigned inpw(*port*);	Reads a word
unsigned *port*;	Port number

Description

The **inp** and **inpw** functions read a byte and a word, respectively, from the specified input port. The input value can be any unsigned integer in the range 0 – 65,535.

Return Value

The functions return the byte or word read from *port*. There is no error return.

See Also

outp, **outpw**

Example

```
#include <conio.h>
#include <stdio.h>

/* Read will be done on port #0: */
unsigned int port = 0;
char result;

main()
 {
  /* Input a byte from the port: */
  result = inp(port);
  printf("The value from port #%d is %d\n", port, result);
 }
```

This program reads a character from input port 0.

Summary

#include <dos.h>

int int86(*intno, inregs, outregs*);	
int *intno*;	Interrupt number
union REGS **inregs*;	Register values on call
union REGS **outregs*;	Register values on return

Description

The **int86** function executes the 8086-processor-family interrupt specified by the interrupt number *intno*. Before executing the interrupt, **int86** copies the contents of *inregs* to the corresponding registers. After the interrupt returns, the function copies the current register values to *outregs*. It also copies the status of the system carry flag to the **cflag** field in *outregs*. The *inregs* and *outregs* arguments are unions of type REGS. The union type is defined in the include file **dos.h**.

The **int86** function is used to invoke MS-DOS interrupts directly.

Return Value

The return value is the value in the **AX** register after the interrupt returns. If the **cflag** field in *outregs* is nonzero, an error has occurred; in such cases, the **_doserrno** variable is also set to the corresponding error code.

See Also

bdos, **intdos**, **intdosx**, **int86x**

Example

```
#define VIDEO_IO  0x10
#define SET_CRSR  1

#include <dos.h>
#include <stdio.h>

union REGS regs;

main()
 {
  int top, bot;

  /* Get new cursor size from user: */
  printf ("Enter new cursor top and bottom: ");
  scanf("%d %d", &top, &bot);

  /* Set up for cursor change call: */
  regs.h.ah = SET_CRSR;
  regs.h.ch = top;
  regs.h.cl = bot;

  /* Execute interrupt: */
  int86(VIDEO_IO, &regs, &regs);
 }
```

This program uses **int86** to call the IBM-PC BIOS video service (INT 10H) to change the size of the cursor.

The default values are as follows:

Configuration	Default Values
Monochrome card	12, 13
Color card	6, 7
43-line EGA	4, 5

Summary

```
#include <dos.h>

int int86x(intno, inregs, outregs, segregs);
int intno;                                     Interrupt number
union REGS {
        struct WORDREGS {
                unsigned int ax;
                unsigned int bx;
                unsigned int cx;
                unsigned int dx;
                unsigned int si;
                unsigned int di;
                unsigned int cflag;
                } x;
        struct BYTEREGS {
                unsigned char al, ah;
                unsigned char bl, bh;
                unsigned char cl, ch;
                unsigned char dl, dh;
                } h;
        } *inregs;                             Register values on call
union REGS *outregs;                           Register values on return
struct SREGS {
        unsigned int es;
        unsigned int cs;
        unsigned int ss;
        unsigned int ds;
        } *segregs;                            Segment-register values on call
```

Description

The **int86x** function executes the 8086-processor-family interrupt specified by the interrupt number *intno*. Unlike the **int86** function, **int86x** accepts segment-register values in *segregs*, letting programs that use large-model data segments or far pointers specify which segment or pointer should be used during the system call.

Before executing the specified interrupt, **int86x** copies the contents of *inregs* and *segregs* to the corresponding registers. Only the **DS** and **ES** register values in *segregs* are used. After the interrupt returns, the function copies the current register values to *outregs*, copies the current **ES** and **DS** values to *segregs*, and restores **DS**. It also copies the status of the system carry flag to the **cflag** field in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The *segregs* argument is a structure of type **SREGS**. These types are defined in the include file **dos.h**.

The **int86x** function is used to directly invoke MS-DOS interrupts that take an argument in the **ES** register, or that take a **DS** register value different from the default data segment.

Return Value

The return value is the value in the **AX** register after the interrupt returns. If the **flag** field in *outregs* is nonzero, an error has occurred; in such cases, the **doserrno** variable is also set to the corresponding error code.

See Also

bdos, **FP_SEG**, **intdos**, **intdosx**, **int86**, **segread**

Note

Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

Example

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

#define SYSCALL     0x21  /* INT 21H invokes system calls */
#define CHANGE_ATTR 0x43  /* System call 43H */
                          /* actually changes attributes */

char far *filename = "int86x.c";  /* filename in 'far' */
                                  /* data segment */
union REGS inregs, outregs;
struct SREGS segregs;
int result;
```

```
main()
 {
  /*
  ** AH us system call number
  ** AL is function (get attributes)
  ** DS:DX points to file name
  */

  inregs.h.ah = CHANGE_ATTR;
  inregs.h.al = 0;
  inregs.x.dx = FP_OFF(filename);
  segregs.ds  = FP_SEG(filename);
  result = int86x(SYSCALL, &inregs, &outregs, &segregs);
  if (outregs.x.cflag)
   {
    printf( "Can't get file attributes; error no. %d\n",
            result);
    exit(1);
   }
  else
    printf("Attribs = %#x\n", outregs.x.cx );
  }
```

In this program, **int86x** executes an INT 21H instruction to invoke MS-DOS system call 43H (change file attributes). The program uses **int86x** because the file, which is referenced with a **far** pointer, may be in a segment other than the default data segment. Thus, the program must explicitly set the **DS** register with the **SREGS** structure.

■ Summary

#include <dos.h>

int intdos(*inregs, outregs*);	
union REGS **inregs*;	Register values on call
union REGS **outregs*;	Register values on return

■ Description

The **intdos** function invokes the MS-DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The union type is defined in the include file **dos.h**.

To invoke a system call, **intdos** executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* to the corresponding registers. After the INT instruction returns, **intdos** copies the current register values to *outregs*. It also copies the status of the system carry flag to the **cflag** field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

The **intdos** function is used to invoke MS-DOS system calls that take arguments in registers other than **DX** (**DH**/**DL**) and **AL**, or to invoke system calls that indicate errors by setting the carry flag.

■ Return Value

The **intdos** function returns the value of the **AX** register after the system call is completed. If the **cflag** field in *outregs* is nonzero, an error has occurred and **_doserrno** is also set to the corresponding error code.

■ See Also

bdos, **intdosx**

Example

```
#include <dos.h>
#include <stdio.h>

union REGS inregs, outregs;

main()
 {

  /* Setup for function call 2a hex: */
  inregs.h.ah = 0x2a;

  /* Get current date:              */
  intdos(&inregs,&outregs);
  printf("date is %d/%d/%d\n",
         outregs.h.dh,outregs.h.dl,outregs.x.cx);
 }
```

This program uses **intdos** to invoke MS-DOS system call 2AH (get the current date).

intdosx

■ Summary

#include <dos.h>

int intdosx(*inregs***,** *outregs***,** *segregs***);**	
union REGS ****inregs*;**	Register values on call
union REGS ****outregs*;**	Register values on return
struct SREGS ****segregs*;**	Segment-register values on call

■ Description

The **intdosx** function invokes the MS-DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The **REGS** and **SREGS** unions are described in the reference page for **int86x**. Unlike the **intdos** function, **intdosx** accepts segment-register values in *segregs*, letting programs that use long-model data segments or far pointers specify which segment or pointer should be used during the system call. The *inregs* and *outregs* arguments are unions of type **REGS**. The *segregs* argument is a structure of type **SREGS**. These types are defined in the include file **dos.h**.

To invoke a system call, **intdosx** executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* and *segregs* to the corresponding registers. Only the **DS** and **ES** register values in *segregs* are used. After the INT instruction returns, **intdosx** copies the current register values to *outregs* and restores **DS**. It also copies the status of the system carry flag to the **cflag** field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

The **intdosx** function is used to invoke MS-DOS system calls that take an argument in the **ES** register, or that take a **DS** register value different from the default data segment.

■ Return Value

The **intdosx** function returns the value of the **AX** register after the system call is completed. If the **cflag** field in *outregs* is nonzero, an error has occurred; in such cases, **_doserrno** is also set to the corresponding error code.

See Also

bdos, **FP_SEG** **intdos**, **segread**,

Note

> Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

Example

```
#include <dos.h>
#include <stdio.h>
#include <direct.h>

union REGS inregs, outregs;
struct SREGS segregs;
char buffer[51], buf2[51];    /* Buffers for directory names */
char far *dir = "\newdir";    /* Directory to create */
char *result1, result2;

main()
{
 result1 = getcwd(buffer,50);
 printf("Current working directory is =%s\n", buffer);

 mkdir(dir);
 inregs.h.ah = 0x3b;           /* Change directory function */
 inregs.x.dx = FP_OFF(dir);    /* File name offset */
 segregs.ds = FP_SEG(dir);     /* File name segment */
 intdosx(&inregs,&outregs,&segregs);
 result1 = getcwd(buf2,50);
 printf("Changed working directory is =%s\n", buf2 );
 result2 = chdir(buffer);      /* Change back */
 result1 = getcwd(buf2, 50);
 printf("Changed to original working directory =%s\n", buf2);
}
```

First, this program gets and displays the name of the current directory and creates a directory named `\newdir` on the current drive. Then it invokes MS-DOS system call 3BH (change directory) using **intdosx** to change the current working directory to `\newdir`. Finally, it restores the original directory as the current working directory.

isalnum – isascii

Summary

#include <ctype.h>

int isalnum(*c***);**	Tests for alphanumeric ('A'–'Z', 'a'–'z', or '0'–'9')
int isalpha(*c***);**	Tests for letter ('A'–'Z' or 'a'–'z')
int isascii(*c***);**	Tests for ASCII character (0x00–0x7F)
int *c***;**	Integer to be tested

Description

The **ctype** routines listed above test a given integer value, returning a nonzero value if the integer satisfies the test condition and a 0 value if it does not. An ASCII-character-set environment is assumed.

The **isascii** routine produces meaningful results for all integer values. However, the remaining routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where **isascii** holds true) or for the non-ASCII value **EOF** (defined in **stdio.h**).

See Also

iscntrl, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, **toascii**, **tolower**, **toupper**

Note

The **ctype** routines are implemented as macros.

Example

```
#include <stdio.h>
#include <ctype.h>

main()
 {
  int ch;
  for (ch = 0; ch <= 0x7f; ch++)
   {
    printf("%#04x",ch);
    printf("%3s",isalnum(ch) ? "AN" : "");
    printf("%2s",isalpha(ch) ? "A"  : "");
    printf("%3s",isascii(ch) ? "AS" : "");

    putchar('\n');
   }
 }
```

This program uses **isalnum**, **isalpha**, and **isascii** to test all characters between 0x0 and 0x7F. It displays each character tested, followed by a code indicating the character type: A for alpha characters, AN for alphanumeric characters, and AS for ASCII characters.

isatty

Summary

#include <io.h>	Required only for function declarations
int isatty(*handle*);	
int *handle*;	Handle referring to device to be tested

Description

The **isatty** function determines whether *handle* is associated with a character device (that is, a terminal, console, printer, or serial port).

Return Value

The **isatty** function returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

Example

```
#include <stdio.h>
#include <io.h>

long loc;

main()
{
 int interactive;

 interactive = isatty(fileno(stdout));
 printf("Is stdout redirected? %s0, interactive ? "no" : "yes");
 /* if not a character device, get current position */
 if (!interactive)
  loc = tell(fileno(stdout));
}
```

This program checks to see whether **stdout** has been redirected to a file. For example, if the program was invoked as

```
sample > output
```

then

```
isatty(fileno(stdout))
```

would return *false* because **stdout** is actually the file output. If the program is invoked as

```
sample
```

however, then the call to **isatty** would return *true*, because **stdout** is still directed to the screen.

iscntrl – isxdigit

Summary

#include <ctype.h>

int iscntrl(*c***);**	Tests for control character (0x00–0x1f or 0x7f)
int isdigit(*c***);**	Tests for digit (’0’–’9’)
int isgraph(*c***);**	Tests for printable character not including the space character (0x21–0x7e)
int islower(*c***);**	Tests for lowercase (’a’–’z’)
int isprint(*c***);**	Tests for printable character (0x20–0x7e)
int ispunct(*c***);**	Tests for punctuation character
int isspace(*c***);**	Tests for white-space character (0x09–0x0d or 0x20)
int isupper(*c***);**	Tests for uppercase (’A’–’Z’)
int isxdigit(*c***);**	Tests for hexadecimal digit (’A’–’F’,’a’–’f’,or ’0’–’9’)
int *c***;**	Integer value to be tested

Description

The **ctype** macros listed above test a given integer value and return a nonzero value if the integer satisfies the test condition, and 0 if it does not. An ASCII-character-set environment is assumed.

These routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where **isascii** holds true) or for the non-ASCII value **EOF** (defined in **stdio.h**).

Return Value

All of these functions return a nonzero value if the tested character is in the right category, and a 0 if not.

See Also

isalnum, **isalpha**, **isascii**, **toascii**, **tolower**, **toupper**

Note

The **ctype** routines are implemented as macros.

Example

```
#include <stdio.h>
#include <ctype.h>

main()
 {
  int ch;
  for (ch = 0; ch <= 0x7f; ch++)
   {
    printf("%2s",iscntrl(ch)  ? "C"  : "");
    printf("%2s",isdigit(ch)  ? "D"  : "");
    printf("%2s",isgraph(ch)  ? "G"  : "");
    printf("%2s",islower(ch)  ? "L"  : "");
    printf(" %c",isprint(ch)  ? ch   : '\0');
    printf("%3s",ispunct(ch)  ? "PU" : "");
    printf("%2s",isspace(ch)  ? "S"  : "");
    printf("%3s",isprint(ch)  ? "PR" : "");
    printf("%2s",isupper(ch)  ? "U"  : "");
    printf("%2s",isxdigit(ch) ? "X"  : "");

    putchar('\n');
   }
 }
```

This program tests all characters between 0x0 and 0x7f, then displays each character with any of the following character-type codes that apply:

Code	Type
C	Control
D	Digit
G	Graphics
L	Lowercase

`PR`	Printable
`S`	Space
`PU`	Punctuation
`U`	Uppercase
`X`	Hexadecimal digit

The program prints all printable characters in the tested range.

Summary

# include <stdlib.h>	Required only for function declarations
char *itoa(*value, string, radix***);**	
int *value***;**	Number to be converted
char ***string;**	String result
int *radix***;**	Base of *value*

Description

The **itoa** function converts the digits of the given *value* to a null-terminated character string and stores the result (up to 17 bytes) in *string*. The *radix* argument specifies the base of *value;* it must be in the range 2–36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–).

Return Value

The **itoa** function returns a pointer to *string*. There is no error return.

See Also

ltoa, **ultoa**

Example

```
#include <stdlib.h>
#include <stdio.h>

int radix = 8;
char buffer[20];
char *p;
main()
 {
  p = itoa(-3445,buffer,radix);        /* p = "171213" */
  printf( "buffer= \"%s\"\n", buffer );
 }
```

This program displays –3445 as a base-8 string.

kbhit

Summary

#include <conio.h> Required only for function declarations

int kbhit(void);

Description

The **kbhit** function checks the console for a recent keystroke.

Return Value

The **kbhit** function returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Example

```
main()
 {
  printf( "waiting...\n" );

  /* Loop until kbhit() reports a keystroke: */
  while( !kbhit() );

  printf( "key struck was '%c'\n", getch() );
 }
```

This program loops until the user presses a key. If **kbhit** returns nonzero, a keystroke is waiting in the buffer. The program can call **getch** or **getche** to fetch the keystroke. If the program calls either function without first checking **kbhit**, the program may pause while waiting for input.

Summary

#include <stdlib.h>	Required only for function declarations
long labs(*n*);	
long *n*;	Long integer value

Description

The **labs** function produces the absolute value of its long-integer argument *n*.

Return Value

The **labs** function returns the absolute value of its argument. There is no error return.

See Also

abs, **cabs**, **fabs**

Example

```
#include <stdlib.h>
#include <stdio.h>

main()
 {
  long x,y;

  x = -41567L;
  y = labs(x);
  printf("The labs(%ld) = %ld",x,y);
 }
```

This program uses **labs** to get and display the absolute value of –41,567.

ldexp

Summary

#include <math.h>

double ldexp(*x*, *exp*);	
double *x*;	Floating-point value
int *exp*;	Integer exponent

Description

The **ldexp** function calculates the value of $x * 2^{exp}$.

Return Value

The **ldexp** function returns $x * 2^{exp}$. If an overflow results, the function returns ±**HUGE_VAL** (depending on the sign of x) and sets **errno** to **ERANGE**.

See Also

frexp, **modf**

Example

```
#include <math.h>

main()
 {
  double x,y;
  int p;
  x = 1.5;
  p = 5;
  y = ldexp(x,p);                         /* y = 48.0 */
  printf("The ldexp(%f,%d) = %f",x,p,y);
 }
```

This program uses **ldexp** to calculate the value of $1.5 * 2^{5}$.

Summary

#include <stdlib.h>

struct ldiv_t {	
long int quot;	Quotient
long int rem;	Remainder
} ldiv(*numer*, *denom*);	
long int *numer*;	Numerator
long int *denom*;	Denominator

Description

The **ldiv** function divides *numer* by *denom*, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient. If the denominator is 0 the program will terminate with an error message.

The **ldiv** function is similar to the **div** function, the difference being that the arguments and the members of the returned structure are all of type **long int**.

Return Value

The **ldiv** function returns a structure of type **ldiv_t**, comprising both the quotient and the remainder. The structure is defined in **stdlib.h**.

See Also

div

Example

```
#include <stdlib.h>
#include <math.h>

main(argc, argv)
int argc;
char **argv;
 {
  long int x,y;
  ldiv_t div_result;

  x = atol(argv[1]);
  y = atol(argv[2]);
  printf("x is %ld, y is %ld\n", x,y);

  div_result = ldiv(x,y);
  printf("The quotient is %ld, and the remainder is %ld\n",
         div_result.quot, div_result.rem);

 }
```

This program takes two long integers as command-line arguments and displays the results of the integer division.

Summary

# include <search.h>	Required only for function declarations

char *lfind(*key, base, num, width,* (*compare*)());

char *lsearch(*key, base, num, width,* (*compare*)());

**char ****key*;	Object to search for
**char ****base*;	Pointer to base of search data
**unsigned ****num, width*;	Number and width of elements
int (**compare*)(*elem1, elem2*);	Pointer to compare function
**const void ****elem1*, ******elem2*;	Array elements to compare

Description

The **lsearch** and **lfind** functions perform a linear search for the value *key* in an array of *num* elements, each of *width* bytes in size. (Unlike **bsearch**, **lsearch** and **lfind** do not require the array to be sorted.) The argument *base* is a pointer to the base of the array to be searched.

If *key* is not found, **lsearch** adds it to the end. The **lfind** function does not.

The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. Both **lsearch** and **lfind** call the *compare* routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements, then return one of the following values:

Value	Meaning
Nonzero	*element1* and *element2* are different
0	*element1* is identical to *element2*

Return Value

If the key is found, both **lsearch** and **lfind** return a pointer to the array element *base* that matches *key*. If the key is not found, **lfind** returns **NULL**, and **lsearch** returns a pointer to a newly added item at the end of the array.

See Also

bsearch

Example

```
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare();             /* must declare as a function */

main (argc, argv)
int argc;
char **argv;
 {
  char **result;
  char *key = "PATH";

  result = (char **)lfind((char *)&key, (char *)argv, &argc,
                             sizeof(char *), compare);
  if (result)
    printf("%s found\n", *result);
   else
     printf("PATH not found!\n");
 }

int compare (arg1, arg2)
char **arg1, **arg2;
 {
  return(strncmp(*arg1,*arg2,strlen(*arg1)));
 }
```

This program uses **lfind** to search for the key word PATH in the command-line arguments. Unlike **lsearch**, **lfind** fails if the key word is not found.

Summary

#include <graph.h>

short far _lineto(*x*, *y*);
short *x*, *y*; End point

Description

The **_lineto** function draws a line from the current position up to and including the logical point (*x*, *y*). The line is drawn using the current color and line style. If no error occurs, **_lineto** sets the current position to the logical point (*x*, *y*).

Note

> If you use **_floodfill** to fill in a closed figure drawn with **_lineto** calls, the figure must be drawn with a solid line-style pattern.

Return Value

The **_lineto** function returns a nonzero value if the line is drawn successfully; otherwise, it returns 0.

See Also

_getcurrentposition, **_setlinestyle**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop;
 _setvideomode(_MRES16COLOR);
 _moveto( 80, 50 );
 _lineto( 240, 150 );
 _lineto( 240, 50 );
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws the figure shown in Figure R.3.

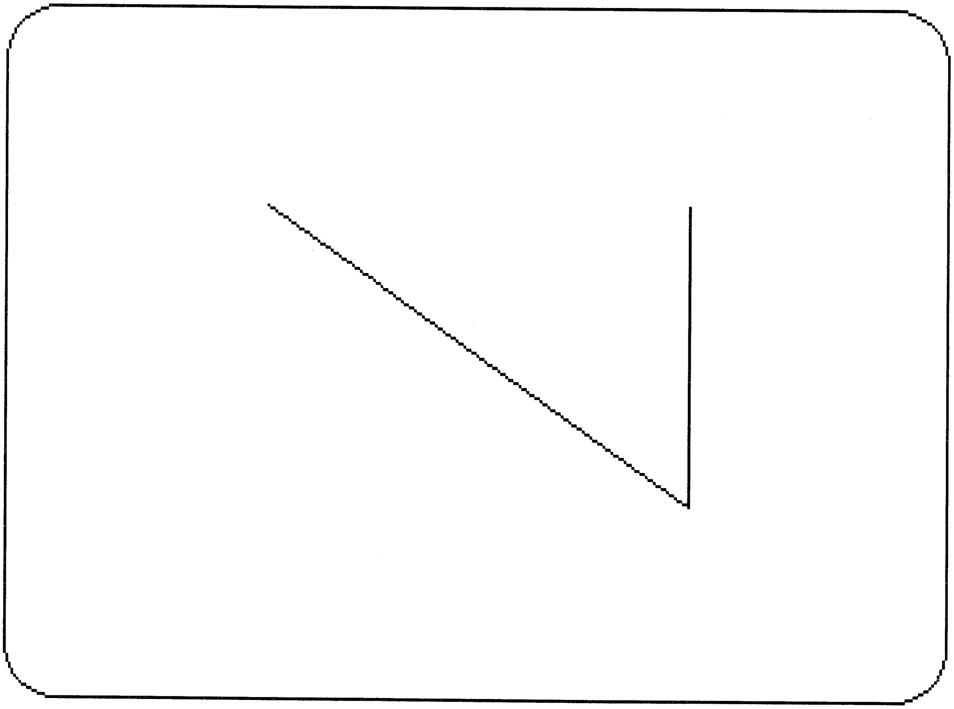


Figure R.3 Output of _lineto Program

Summary

#include <time.h>

struct tm *localtime(*time*);	
**time_t ****time*;	Pointer to stored time

Description

The **localtime** function converts a time stored as a **long** value to a structure. The **long** value *time* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time; this value is usually obtained from the **time** function.

The **localtime** function breaks down the *time* value, corrects for the local time zone and daylight saving time if appropriate, and stores the corrected time in a structure of type **tm**.

The fields of the structure type **tm** store the following values:

Field	Value Stored
tm_sec	Seconds
tm_min	Minutes
tm_hour	Hours (0–24)
tm_mday	Day of month (1–31)
tm_mon	Month (0–11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0–6; Sunday = 0)
tm_yday	Day of year (0–365; January 1 = 0)
tm_isdst	Nonzero if daylight saving time is in effect, otherwise 0

The complete structure is shown in the reference page for **asctime**.

The **localtime** function makes corrections for the local time zone if the user first sets the environment variable **TZ**. The value of **TZ** must be a three-letter time zone name (such as PST), followed by a signed or unsigned number giving the difference between Greenwich mean time and the local time zone. The number may be followed by a three-letter daylight-saving-time zone (such as PDT). The **localtime** function uses the difference between Greenwich mean time and local time to adjust the stored time value. If a daylight-saving-time zone is present in the **TZ** setting, **localtime** also corrects for daylight saving time. If **TZ** currently has no value, the default value PST8PDT is used.

When **TZ** is set, three other environment variables, **timezone**, **daylight**, and **tzname**, are automatically set as well. See the **tzset** function for a description of these variables.

Note

The **TZ** variable is not part of the ANSI standard definition of **localtime**, but is a Microsoft extension.

Return Value

The **localtime** function returns a pointer to the structure result. MS-DOS doesn't understand dates prior to 1980. If *time* is prior to January 1, 1980, the function returns **NULL**.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, if *time* represents a date before 1980, **localtime** returns the structure representation of 00:00:00 January 1, 1980.

See Also

asctime, **ctime**, **ftime**, **gmtime**, **time**, **tzset**

Note

The **gmtime** and **localtime** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

■ Example

```
#include <stdio.h>
#include <time.h>

struct tm *newtime;
long ltime;

main()
 {
  struct tm *newtime;
  char *am_pm = "PM";
  time_t long_time;

  time(&long_time);                /* Get time as long integer */
  newtime = localtime(&long_time); /* Convert to local time */

  if (newtime->tm_hour < 12)       /* Set up extension */
    am_pm = "AM";
  if (newtime->tm_hour > 12)       /* Convert from 24-hour */
    newtime->tm_hour -=12;         /* to 12-hour clock */

  printf("%.19s %s\n",asctime(newtime), am_pm);
 }
```

Sample output:

```
Tue Dec 10 11:30:12 AM
```

This program uses **time** to get the current time and then uses **localtime** to convert this time to a structure representing the local time. The program converts the result from a 24-hour clock to a 12-hour clock and determines the proper extension (AM or PM).

locking

Summary

#include <sys\locking.h>	
#include <io.h>	Required only for function declarations
int locking(*handle, mode, nbyte***);**	
int *handle***;**	File handle
int *mode***;**	File locking mode
long *nbyte***;**	Number of bytes to lock

Description

The **locking** function locks or unlocks *nbyte* bytes of the file specified by *handle*. Locking bytes in a file prevents subsequent reading and writing of those bytes by other processes. Unlocking a file permits other processes to read or write to previously locked bytes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbyte* bytes, or to the end of the file.

Important

Under MS-DOS Versions 3.0 and 3.1, the files locked by a parent process may become unlocked when one of its children exits.

The argument *mode* specifies the locking action to be performed. It must be one of the following manifest constants:

Constant	Action
LK_LOCK	Locks the specified bytes. If the bytes cannot be locked, tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, returns an error.
LK_RLCK	Same as **LK_LOCK**.
LK_NBLCK	Locks the specified bytes. If bytes cannot be locked, returns an error.
LK_NBRLCK	Same as **LK_NBLCK**.
LK_UNLCK	Unlocks the specified bytes. (The bytes must have been previously locked.)

More than one region of a file can be locked, but no overlapping regions are allowed. Furthermore, no more than one region can be unlocked at a time.

When unlocking a file, the region of the file being unlocked must correspond to a region that was previously locked. The **locking** function does not merge adjacent regions, so if two locked regions are adjacent, each region must be unlocked separately.

All locks should be removed before closing a file or exiting the program.

Return Value

The **locking** function returns 0 if it is successful. A return value of –1 indicates failure, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Locking violation (file already locked or unlocked).
EBADF	Invalid file handle.
EDEADLOCK	Locking violation. This is returned when the **LK_LOCK** or **LK_RLCK** flag is specified and the file cannot be locked after 10 attempts.
EINVAL	An invalid argument was given to the function.

See Also

creat, open

Note

The **locking** function should be used only under MS-DOS Versions 3.0 and later; it has no effect under earlier versions of MS-DOS.

Example

```
#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys\locking.h>
#include <stdlib.h>

extern unsigned char _osmajor;
int fh;
long pos;
char buffer[ BUFSIZ ];

main()
 {
  int result;

  /* save the current file pointer position,
  ** then lock a region from the beginning of
  ** the file to the saved file pointer
  ** position
  */

  /* Open file; read 10 bytes: */
  fh = open("data", O_RDONLY);
  result = read(fh, buffer, 10) ;
  if (_osmajor >= 3) {     /* Check for DOS version >= 3.0 */
    pos = tell(fh);        /* Get current pointer position*/
    /* Reset pointer to beginning of file: */
    result = lseek(fh, 0L, SEEK_SET);
    /* Lock first portion of the file: */
    if ((locking(fh, LK_NBLCK, pos)) != -1) {
      printf("Succesfully locked %d bytes\n", pos);
      lseek(fh, 0L, 0 );
      locking(fh, LK_UNLCK, pos);
     }
    else
      perror("Locking failed");
   }
  else
    printf( "MS-DOS version must be 3, or higher.\n" );
 }
```

This program opens a file named `data` and reads the first 10 bytes from the file. It then moves the file pointer back to the beginning of the file and uses **locking** to lock the first 10 bytes of the file.

Note that this program works correctly only if the following conditions are met:

- The file named `data` exists.
- **SHARE.COM** or **SHARE.EXE** is installed.
- The program is run under MS-DOS Version 3.0 or later.

log, log10

Summary

#include <math.h>

double log(*x*);	Calculates natural logarithm of *x*
double log10(*x*);	Calculates base-10 logarithm of *x*
double *x*;	Floating-point value

Description

The **log** and **log10** functions calculate the natural logarithm and base-10 logarithm of *x*, respectively.

Return Value

The **log** and **log10** functions return the logarithm result. If *x* is negative, both functions print a **DOMAIN** error message to **stderr**, return the value –**HUGE_VAL**, and set **errno** to **EDOM**. If *x* is 0, both functions print a **SING** error message to **stderr**, return the value –**HUGE_VAL**, and set **errno** to **ERANGE**.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, both **log** and **log10** set **errno** to **EDOM** whether *x* was 0 or a negative value.

Error handling can be modified by using the **matherr** routine.

See Also

exp, **matherr**, **pow**

■ Example

```
#include <math.h>
#include <stdio.h>

main()
 {
  double x = 1000.0, y;

  y = log(x);                                /* y is 6.907755 */
  printf("The log(%.2f) = %f\n",x,y);

  y = log10(x);                              /* y is 3.0 */
  printf("The log10(%.2f) = %f\n",x,y);
 }
```

This program uses **log** and **log10** to calculate the natural logarithm and the base-10 logarithm of 1000, respectively.

longjmp

Summary

#include <setjmp.h>

void longjmp(*env, value***);**

jmp_buf *env*;	Variable in which environment is stored
int *value*;	Value to be returned to **setjmp** call

Description

The **longjmp** function restores a stack environment previously saved in *env* by **setjmp**. The **setjmp** and **longjmp** functions provide a way to execute a nonlocal goto and are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point immediately following the corresponding **setjmp** call. Execution resumes as if *value* had just been returned by the **setjmp** call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable.

The **longjmp** function must be called before the function that called **setjmp** returns. If **longjmp** is called after the function calling **setjmp** returns, unpredictable program behavior will result.

The value returned by **longjmp** must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.

Return Value

There is no return value.

See Also

setjmp

Warning

The values of register variables in the routine calling **setjmp** may not be restored to the proper values after a **longjmp** is executed.

Example

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

main()
 {
  if (setjmp(mark) != 0) {  /* Set the point to jump to */
    printf("longjmp has been called\n");
    recover();
    exit(1);
   }
  printf("setjmp has been called\n");
  p();
 }

p()
 {                    /* Routine to trigger an error */
  int error = 0;
  error = 1;
  if (error != 0)
    longjmp(mark,-1);         /* Execute a long jump */
  }

recover()        /* Code goes here for recovery */
 {               /* from the error              */
 /* Exit the program to ensure that data files */
 /* won't be corrupted                         */
 }
```

This program uses **setjmp** to save the stack environment and executes the p function to simulate an error. It then uses **longjmp** to restore the stack environment and resume execution immediately after the **setjmp** call. Because **longjmp** and **setjmp** return different values, a conditional expression in the program allows the program to call the `recover` function to use additional error-recovery code.

_lrotl, _lrotr

Summary

#include <stdlib.h>

unsigned long _lrotl(*value, shift***);**	Rotates left
unsigned long _lrotr(*value, shift***);**	Rotates right
unsigned long *value***;**	Value to be rotated
int *shift***;**	Number of bits to shift

Description

The **_lrotl** and **_lrotr** functions rotate *value* by *shift* bits.

Return Value

Both functions return the rotated value. There is no error return.

See Also

_rotl, **_rotr**

Example

```
#include <stdlib.h>

main()
{
 unsigned long val = 0X01234567;
 printf("_lrotl(val,4)  = 0x%8.8lx\n", _lrotl(val,4));
 printf("_lrotr(val,16) = 0x%8.8lx\n", _lrotr(val,16));
}
```

The output would look like this:

```
_lrotl(val,4)  = 0x12345670
_lrotr(val,16) = 0x45670123
```

This program uses **_lrotl** and **_lrotr** with different *shift* values to rotate the long integer value `0x1234567`.

Summary

#include <io.h>	Required only for function declarations
#include <stdio.h>	
long lseek(*handle, offset, origin***);**	
int *handle***;**	Handle referring to open file
long *offset***;**	Number of bytes from *origin*
int *origin***;**	Initial position

Description

The **lseek** function moves the file pointer (if any) associated with *handle* to a new location that is *offset* bytes from *origin.* The next operation on the file occurs at the new location. The *origin* must be one of the following constants defined in **stdio.h**:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

The **lseek** function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However an attempt to position the pointer before the beginning of the file causes an error.

Return Value

The **lseek** function returns the offset, in bytes, of the new position from the beginning of the file. A return value of –1L indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EBADF	Invalid file handle
EINVAL	Invalid value for *origin*, or position specified by *offset* is before the beginning of the file

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

See Also

fseek, **tell**

Example

```
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int fh;
long pos;          /* position of file pointer */
char buffer[10];

main()
 {
  fh = open("data",O_RDONLY);


  /* Seek the beginning of the file: */
  pos = lseek(fh,0,SEEK_SET);
  if (pos == -1L)
    perror("lseek to beginning failed");
  else
    printf("Position for beginning of file seek = %d\n", pos);

  read(fh,buffer,10);           /* Move file pointer a little */

  /* Find current position: */
  pos = lseek(fh,0L,SEEK_CUR);
  if (pos == -1L)
    perror("lseek to current position failed");
  else
    printf("Position for current position seek = %d\n", pos);

  /* Set the end of the file: */
  pos = lseek(fh,0L,SEEK_END);
  if (pos == -1L)
    perror("lseek to end failed");
  else
    printf("Position for end of file seek = %d\n", pos);
}
```

This program first opens a file named `data`. It then uses **lseek** to find the beginning of the file, to find the current position in the file, and to find the end of the file.

ltoa

Summary

#include <stdlib.h>	Required only for function declarations
char *ltoa(*value, string, radix***);**	
long *value***;**	Number to be converted
char ****string*;**	String result
int *radix***;**	Base of *value*

Description

The **ltoa** function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *string.* The *radix* argument specifies the base of *value*; it must be in the range 2–36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (–).

Return Value

The **ltoa** function returns a pointer to *string.* There is no error return.

See Also

itoa, **ultoa**

Example

```
#include <stdlib.h>

int radix = 10;
char buffer[20];
char *p;
main()
{
 p = ltoa(-344115L,buffer,radix); /* p = "-344115" */
 printf( "Buffer= \"%s\"\n", buffer );
}
```

This program displays the long integer –344,115 as a base-10 string.

Summary

#include <stdlib.h>

void _makepath(*path, drive, dir, fname, ext***);**

char **path*;	Full path-name buffer
char **drive*;	Drive letter
char **dir*;	Directory path
char **fname*;	File name
char **ext*;	File extension

Description

The **_makepath** routine creates a single path name, composed of a drive letter, directory path, file name, and file-name extension. The *path* argument should point to an empty buffer large enough to hold the complete path name. The constant **_MAX_PATH**, defined in **stdlib.h**, specifies the maximum size *path* that MS-DOS can handle. The other arguments point to the following buffers containing the path-name elements:

Buffer	Description
drive	The *drive* argument contains a letter (A, B, etc.) corresponding to the desired drive and an optional trailing colon. The **_makepath** routine will insert the colon automatically in the composite path name if it is missing. If *drive* is a null character or an empty string, no drive letter and colon will appear in the composite *path* string.
dir	The *directory* argument contains the path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either forward slashes (/) or backslashes (\) or both may be used in a single *dir* argument. If a trailing slash (/ or \) is not specified, it will be inserted automatically. If *dir* is a null character or an empty string, no slash is inserted in the composite *path* string.
fname	The *fname* argument contains the base file name without any extensions.
ext	The *ext* argument contains the actual filename extension, with or without a leading period (.). The **_makepath** routine will insert the period automatically if it doesn't appear in *ext*. If *ext* is a null character or an empty string, no period is inserted in the composite *path* string.

There are no size limits on any of the above four fields. However, the composite path should be no larger than the **_MAX_PATH** constant; otherwise, MS-DOS will not handle it correctly.

Example

```
#include <dos.h>

main()
 {
  char path_buffer [40];
  char * drive [3];
  char * dir [30];
  char * fname [9];
  char * ext [4];

  _makepath (path_buffer, "c", "qc\\clibref\\", "makepath", "c");
  printf ("path created with _makepath: %s\n\n", path_buffer);

  _splitpath (path_buffer, drive, dir, fname, ext);
  printf ("path extracted with _splitpath\n");
  printf ("drive: %s\n", drive);
  printf ("dir: %s\n", dir);
  printf ("fname: %s\n", fname);
  printf ("ext: %s\n", ext);
 }
```

This program builds a file-name path from the specified components.

Summary

#include <stdlib.h>	For ANSI compatibility (**malloc** only)
#include <malloc.h>	Required only for function declarations
void *malloc(*size*);	Allocates a memory block
size_t *size*;	Bytes to allocate
void far *_fmalloc(*size*);	Allocates a memory block in the far heap
size_t *size*;	Bytes to allocate
void near *_nmalloc(*size*);	Allocates a memory block in the near heap
size_t *size*;	Bytes to allocate

Description

The **malloc** function allocates a memory block of at least *size* bytes. (The block may be larger than *size* bytes because of space required for alignment and for maintenance information.)

If *size* is 0, **malloc** returns **NULL**.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, **malloc** allocates a zero-length item (that is, a header only) in the heap if *size* is 0. The resulting pointer can be passed to the **realloc** function to adjust the size at any time.

In large data models (compact- and large-model programs), **malloc** maps to **_fmalloc**. In small data models (small- and medium-model programs), **malloc** maps to **_nmalloc**.

The **_fmalloc** function allocates a memory block of at least *size* bytes outside the default data segment. (The block may be larger than *size* bytes because of space required for alignment.) The **_fmalloc** function returns a far pointer to **void**. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **char**, use a type cast on the return value.

If sufficient memory is not available outside the default data segment, **_fmalloc** will retry allocating within the default data segment. If there is still insufficient memory available, the return value is **NULL**.

The **_nmalloc** function allocates a memory block of at least *size* bytes inside the default data segment. (The block may be larger than *size* bytes because of space required for alignment.)

Return Value

The **malloc** function returns a **void** pointer to the allocated space, the **_fmalloc** function returns a far pointer to **void**, and the **_nmalloc** function returns a near pointer to **void**.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

The **_malloc** and **_nmalloc** functions return **NULL** if there is insufficient memory available. If **_fmalloc** does not find sufficient memory available outside the default data segment, it will try reallocating inside the default data segment. If there is still insufficient memory available, **_fmalloc** will return **NULL**.

See Also

calloc, **_ffree**, **_fmalloc**, **free**, **malloc**, **_nfree**, **_nmalloc**, **realloc**

Example

```
#include <stdio.h>
#include <malloc.h>

int *intarray;

main()
 {
  /* Allocate space for 20 integers: */
  intarray = (int *)malloc(20*sizeof(int));
  if (intarray == NULL )
    printf( "Insufficient memory available\n" );
  else
    printf( "Memory space allocated for 20 integers.\n" );
 }
```

This program uses **malloc** to allocate space from the heap for 20 integers.

Summary

#include <math.h>

int matherr(*x*);
struct exception { Math exception information:
    int *type*; Error type
    char ****name*; Function where error originates
    double *arg1*, *arg2*; Values that caused the error
    double retval; Return value
    } ****x*;

Description

The **matherr** function processes errors generated by the functions of the math library. The math functions call **matherr** whenever an error is detected. The user can provide a different definition of the **matherr** function to carry out special error handling.

When an error occurs in a math routine, **matherr** is called with a pointer to an **exception** type structure (defined in **math.h**) as an argument. The *type* specifies the type of math error. It will be one of the following values, defined in **math.h**:

Value	Meaning
DOMAIN	Argument domain error
SING	Argument singularity
OVERFLOW	Overflow range error
PLOSS	Partial loss of significance
TLOSS	Total loss of significance
UNDERFLOW	Underflow range error

The structure member *name* is a pointer to a null-terminated string containing the name of the function that caused the error. The structure members *arg1* and *arg2* specify the values that caused the error. (If only one argument is given, it is stored in *arg1*.)

The default return value for the given error is **retval**. If you change the return value, remember that the return value must specify whether an error actually occurred. If **matherr** returns 0, an error message is displayed and **errno** is set to an appropriate error value. If **matherr** returns a nonzero value, no error message is displayed and **errno** remains unchanged.

Return Value

The **matherr** function should return 0 to indicate an error, and nonzero to indicate successful corrective action.

See Also

acos, **asin**, **atan**, **atan2**, **bessel**, **cabs**, **cos**, **cosh**, **exp**, **hypot**, **log**, **pow**, **sin**, **sinh**, **sqrt**, **tan**

Example

```
#include <math.h>
#include <string.h>

main()
 {
  printf("log(-2)=%e, log10(-5)=%e, log(0)=%e\n",
          log(-2.0), log10(-5), log(0));
 }

int matherr(x)
struct exception *x;
 {
  if (x->type == DOMAIN) {  /* If domain error from "log": */
   if (strcmp(x->name, "log") == 0) {
    x->retval = log(-(x->arg1));
    return(1);
   }
   else
    if ( strcmp(x->name, "log10") == 0) { /* If from "log10": */
     x->retval = log10(-(x->arg1));
     return(1);
    }
   }
  return(0);     /* Else use the default actions */
 }
```

If any of the functions this program calls causes an error, the program calls **matherr**. If the error resulted from a negative argument to **log** or **log10** (a domain error), the program returns the natural or base-10 logarithm of the absolute value of the argument and suppresses the usual error message.

Summary

#include <stdlib.h>

type **max(***a, b***);**
*type a, b***;** Values to compare

Description

The **max** macro compares two values and returns the value of the larger one. The data *type* can be any numerical data type, signed or unsigned. The *type* must be the same for both arguments and the function declaration for each call to **max**.

Return Value

The macro returns the larger of the two arguments.

See Also

min

Example

```
#include <stdlib.h>
#include <stdio.h>

main();
 int a = 10;
 int b = 21;

 printf("The larger of %d and %d is %d\n", a, b, (int)max(a, b));
```

This program prints the larger of the two values, a and b.

_memavl

Summary

#include <malloc.h> Required only for function declarations

size_t _memavl(void);

Description

The **_memavl** function returns the approximate size, in bytes, of the memory available for dynamic memory allocation in the default data segment. This function can be used with **calloc**, **malloc**, or **realloc** in the small- and medium-memory models, and with **_nmalloc** in all memory models.

Return Value

The **_memavl** function returns the size in bytes as an unsigned integer.

See Also

calloc, **_freect**, **malloc**, **realloc**, **stackavail**

Example

```
#include <malloc.h>

main()
 {
  long *longptr;

  printf("Memory available before malloc = %u\n", _memavl());
  longptr = (long*)malloc(5000*sizeof(long));
  printf("Memory available after malloc = %u\n", _memavl());
 }
```

Sample output:

```
Memory available before malloc = 61383
Memory available after malloc = 40959
```

This program uses **_memavl** to determine the amount of available memory. It then uses **malloc** to allocate space for 5000 long integers and uses **_memavl** again to determine the new amount of available memory.

memccpy

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**
void *memccpy(*dest***,** *src***,** *c***,** *cnt***);**	
const void ****dest*;**	Pointer to destination
const void ****src*;**	Pointer to source
int *c***;**	Last character to copy
unsigned *cnt***;**	Number of characters

Description

The **memccpy** function copies 0 or more bytes of *src* to *dest*, copying up to and including the first occurrence of the character *c* or until *cnt* bytes have been copied, whichever comes first.

Return Value

If the character *c* is copied, **memccpy** returns a pointer to the byte in *dest* that immediately follows the character. If *c* is not copied, **memccpy** returns **NULL**.

See Also

memchr, **memcmp**, **memcpy**, **memset**

■ Example

```
#include <memory.h>
#include <string.h>
#include <stdio.h>

char buffer[100], source[100] = "This is the "
        " string to be transferred\n";
char *result;

main()
 {
  result = memccpy(buffer, source, '\n',100);

  if (result == NULL)
    printf("Memory has been copied, but no \\n was found");
  else
    printf("Memory has been copied, and a \\n was found");
 }
```

This program uses **memccpy** to copy a string from `source` to `buffer`. The copy proceeds until either 100 bytes have been copied or a new-line character (**\n**) is encountered.

memchr

Summary

# include <memory.h>	Required only for function declarations
# include <string.h>	Use either **string.h** or **memory.h**
void *memchr(_buf, c, count_**);**	
void *_buf_**;**	Pointer to buffer
size_t _c_**;**	Character to copy
unsigned _count_**;**	Number of characters

Description

The **memchr** function looks for *c* in the first *count* bytes of *buf*. It stops when it finds *c* or after checking the first *count* bytes.

Return Value

If successful, **memchr** returns a pointer to the first location of *c* in *buf*. Otherwise, it returns **NULL**.

See Also

memccpy, **memcmp**, **memcpy**, **memset**

Example

```
#include <memory.h>
#include <stdio.h>

char buffer[100];
char *result;
main()
 {
  strcpy(buffer, "this is a test" );
  result = memchr(buffer,'a',100);
  if (result != NULL)
    printf("Char. 'a' found at position %d\n", result-buffer+1);
  else
    printf("Char 'a' not found in first 100 bytes of buffer");
 }
```

This program uses **memchr** to find the first occurrence of a in the buffer.

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**

int memcmp (*buf1*, *buf2*, *count***);**

**const void ****buf1*;	First buffer
**const void ****buf2*;	Second buffer
size_t *count*;	Number of characters

Description

The **memcmp** function compares the first *count* bytes of *buf1* and *buf2* and returns a value indicating their relationship, as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

Note

There is a semantic difference between the function version of **memcmp** and its intrinsic version. The function version supports huge pointers in compact- and large-model programs, but the intrinsic version does not.

Return Value

The **memcmp** function returns an integer value, as described above.

See Also

memccpy, **memchr**, **memcpy**, **memset**

Example

```
#include <string.h>

char first[100], second[100];
int result;

main()
 {
  strcpy(first,  "12345678901234567890" );
  strcpy(second, "12345678901234567891" );
  result = memcmp(first,second,100);
  printf( "First is %s second.\n", result ?
 ( result < 0 ? "less than" : "greater than" ) : "equal to" );
 }
```

This program uses **memcmp** to compare the strings named `first` and `second`. If the first 100 bytes of the strings are equal, the program considers the strings to be equal.

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**
void *memcpy(*dest***,** *src***,** *count***);**	
void ****dest*;**	New buffer
const void ****src*;**	Buffer to copy from
size_t *count***;**	Number of characters to copy

Description

The **memcpy** function copies *count* bytes of *src* to *dest.* If some regions of *src* and *dest* overlap, **memcpy** does *not* ensure that the original *src* bytes in the overlapping region are copied before being overwritten. Use **memmove** to handle overlapping regions.

C 4.0 Difference

In Version 4.0 of the Microsoft C Run-Time Library, **memcpy** does ensure that overlapping regions are copied before being overwritten.

Note

There is a semantic difference between the function version of **memcpy** and its intrinsic version. The function version supports huge pointers in compact- and large-model programs, but the intrinsic version does not.

Return Value

The **memcpy** function returns a pointer to *dest.*

See Also

memccpy, **memchr**, **memcmp**, **memmove**, **memset**

Example

```
#include <memory.h>

char source[200], dest[200];
char *result;

main()
 {
  strcpy(source, "This is the source to be moved." );
  strcpy(dest, "......................................" );

  /* Move 200 bytes from source to dest */
  /*  and return a pointer to dest:     */
  printf("source = %s\ndestination = %s\n\n", source, dest);
  result = memcpy(dest, source,200);
  printf("source = %s\ndestination = %s\nresult = %s\n",
         source, dest, result);
 }
```

This program uses **memcpy** to copy 200 bytes from `source` to `dest` and returns a pointer to `dest`.

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**
int memicmp (*buf1*, *buf2*, *count*);	
void ****buf1*;**	First buffer
void ****buf2*;**	Second buffer
unsigned *count***;**	Number of characters

Description

The **memicmp** function compares the first *count* characters of *buf1* and *buf2* byte-by-byte, without regard to the case of letters in the two buffers; that is, uppercase (capital) and lowercase letters are considered equivalent. All uppercase letters in *buf1* and *buf2* are converted to lowercase before the comparison. The **memicmp** function returns a value indicating the relationship of *buf1* and *buf2*, as follows:

Value	**Meaning**
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

Return Value

The **memicmp** function returns an integer value, as described above.

See Also

memccpy, **memchr**, **memcmp**, **memcpy**, **memset**

Example

```
#include <memory.h>
#include <stdio.h>
#include <string.h>

char first[100], second[100];
int result;

main()
 {
  strcpy(first, "Those Who Will Not Learn from History");
  strcpy(second,"THOSE WHO WILL NOT LEARN FROM their mistakes");
  /* Note that the 29th letter is right here ^ */

  result = memicmp(first,second,29);   /* result is 0 */
  printf("%d\n",result);
 }
```

Output:

```
0
```

This program uses **memicmp** to compare the the first 29 letters of the strings named `first` and `second` without regard to the case of the letters.

Summary

#include <stdlib.h> ANSI version
#include <malloc.h> UNIX System V version

size_t _memmax(void);

Description

The **_memmax** function returns the size (in bytes) of the largest contiguous block of memory that can be allocated from the near heap. Calling **_nmalloc(_memmax())** will succeed so long as **_memmax** returns a nonzero value.

Return Values

The function returns the block size, if successful. Otherwise, it returns 0, indicating that nothing more can be allocated from the near heap.

See Also

malloc, **msize**

Example

```
#include <stddef.h>
#include <malloc.h>
#include <stdio.h>

main()
{
 size_t max = _memmax();
 char near *p;

 if(max){
  p = _nmalloc(max);
  printf(p ? "max allocation succeeded\n" :
         "max allocation failed - should not occur\n");
 }
 else
  printf("near heap is already full\n");
}
```

This program attempts to allocate **_memmax** bytes from the near heap.

memmove

Summary

#include <string.h>

void *memmove(*dest*, *src*, *count*);

void ***dest*;**	Target object
const void ***src*;**	Source object
size_t *count*;	Number of characters to copy

Description

The **memmove** function copies *count* characters from *src* to *dest*. If some regions of *src* and *dest* overlap, **memmove** ensures that the original *src* bytes in the overlapping region are copied before being overwritten.

Return Value

The **memmove** function returns the value of *src*.

See Also

memccpy, **memcpy**

Example

```
#include <stdio.h>
#include <string.h>

char Source[] = ">>>>>>>>>>>>>><<<<<<<<<<<<<<<<";
char Target[] = "We will copy the stuff in here: "
                "and will see if it correctly moves "
                "the string in";
char *ToPrint;

main()
 {
  printf("Target Before: %s\n\n", Target);
  ToPrint = memmove(&Target[32], Source, sizeof(Source));
  printf("Target After:  %s\n", Target);
 }
```

Using **memmove**, the string `Source` is copied into `Target`. Note that the `sizeof` operator gives back the size of the string, including the end-of-string character, effectively shortening `Target`.

memset

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**
void *memset(*dest*, *c*, *count***);**	
**void ****dest*;	Pointer to destination
int *c*;	Character to set
size_t *count*;	Number of characters

Description

The **memset** function sets the first *count* bytes of *dest* to the character *c*. The normal function version of **memset** supports huge pointers in compact- and large-model programs, but the intrinsic version does not.

Return Value

The **memset** function returns a pointer to *dest*.

See Also

memccpy, **memchr**, **memcmp**, **memcpy**

Example

```
#include <memory.h>

char buffer[100];
main()
 {
  char *result;
  result = memset(buffer,'X',20);
  buffer[20] = '\0';
  printf("Buffer = %s", buffer);
 }
```

This program uses **memset** to set the first 20 bytes of `buffer` to X. It then appends a null character (**'\0'**) to the buffer and displays it.

Summary

#include <stdlib.h>

type **min(***a*, *b***);**
type a, *b*; Values to compare

Description

The **min** macro compares two values and returns the value of the smaller one. The data *type* can be any numerical data type, signed or unsigned. The *type* must be the same for both arguments and the function declaration for each call to **min**.

Return Value

The macro returns the smaller of the two arguments.

See Also

max

Example

```
#include <stdlib.h>
#include <stdio.h>

main();
 int a = 10;
 int b = 21;

 printf("The smaller of %d and %d is %d\n", a, b, (int)min(a, b));
```

This program prints the smaller of the two values, `a` and `b`.

mkdir

Summary

#include <direct.h>	Required only for function declarations
int mkdir(*path***);**	
char ****path*;**	Path name for new directory

Description

The **mkdir** function creates a new directory with the specified *path*. Only one directory can be created at a time, so only the last component of *path* can name a new directory.

Return Value

The **mkdir** function returns the value 0 if the new directory was created. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Directory not created. The given name is the name of an existing file, directory, or device.
ENOENT	Path name not found.

See Also

chdir, **rmdir**

Example

```
#include <direct.h>

main()
 {
  int result;

  /* "b:\tmp" could also be used in this call: */
  result = mkdir("b:/tmp");
  if (result == 0)
    printf("Directory 'b:/tmp' was successfully created\n");
  else
    printf("Problem creating directory 'b:/tmp'\n");

  /* "tmp\sub" could also be used: */
  result = mkdir("tmp/sub");
  if (result == 0)
    printf("Directory 'tmp/sub' was successfully created\n");
  else
    printf("Problem creating directory 'tmp/sub'\n");
 }
```

This program uses **mkdir** to create the directories `b:\tmp` and `tmp\sub`.

mktemp

Summary

#include <io.h> Required only for function declarations

char *mktemp(*template***);**
char ****template*;** File-name pattern

Description

The **mktemp** function creates a unique file name by modifying the given *template*. The *template* argument has the form

*base*XXXXXX

where *base* is the part of the new file name supplied by the user and the Xs are placeholders for the part supplied by **mktemp**; **mktemp** preserves *base* and replaces the six trailing X's with an alphanumeric character followed by a five-digit value. The five-digit value is a unique number identifying the calling process. The alphanumeric character is 0 (**'0'**) the first time **mktemp** is called with a given template.

In subsequent calls from the same process with the same template, **mktemp** checks to see if previously returned names have been used to create files. If no file exists for a given name, **mktemp** returns that name. If files exist for all previously returned names, **mktemp** creates a new name by replacing the alphanumeric character in the name with the next available lowercase letter. For example, if the first name returned is `t012345` and this name is used to create a file, the next name returned will be `ta12345`. When creating new names, **mktemp** uses, in order, '0' and then the lowercase letters 'a' to 'z'.

M-O

Return Value

The **mktemp** function returns a pointer to the modified template. The return value is **NULL** if the *template* argument is badly formed or no more unique names can be created from the given template.

See Also

fopen, **getpid**, **open**

Note

The **mktemp** function generates unique file names but does not create or open files.

Example

```
#include <io.h>
#include <stdio.h>

char *template = "fnXXXXXX";
char *result;
char names[5][9];

main()
{
 int i;

 for( i = 0; i < 5; i++) {
  strcpy(names[i], template);
  /* Attempt to find a unique file name: */
  result = mktemp(names[i]);
  if (result == NULL)
   printf("Problem creating the template");
  else {
   printf("Unique file name is %s\n", result);
          fopen(result, "w");
  }
 }
}
```

The above program uses **mktemp** to create five unique file names. It opens each file name to ensure that the next name is unique.

mktime

Summary

#include <time.h>

time_t mktime(*timeptr*);	
**struct tm ****timeptr*;	Local time structure

Description

The **mktime** function converts the local time into a calendar value. The *timeptr* argument points to a structure that contains the local time. The structure is described in the reference page for **asctime**. The converted time has the same encoding as the values returned by the **time** function. The original values of the **tm_wday** and **tm_yday** components of the *timeptr* structure are ignored, and the original values of the other components are not restricted to their normal ranges.

If successful, **mktime** sets the values of **tm_wday** and **tm_yday** appropriately, and sets the other components to represent the specified calendar time, but with their values forced to the normal ranges; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

Note

> MS-DOS does not understand dates prior to 1980. If *timeptr* references a date before January 1, 1980, **mktime** returns –1.

Return Values

The **mktime** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value –1 cast as type **time_t**.

See Also

asctime, gmtime, localtime, time

■ Example

```
#include <time.h>
#include <stdio.h>

struct tm when;
time_t now;
time_t result;
int days;

main()
{
 printf("How many days to look ahead: ");
 scanf("%d", &days);

 time(&now);
 when = *localtime(&now);
 when.tm_mday = when.tm_mday + days;
 if ((result = mktime(&when)) != (time_t)-1)
  printf("\n%d days from now the time will be %s",
         days, asctime(&when));
  else
   perror("mktime failed");
}
```

The example above takes a number of days as input and returns the time, the current date, and the specified number of days.

modf

Summary

# include <math.h>

double modf(*x*, *intptr*);	
double *x*;	Floating-point value
double ****intptr*;	Pointer to stored integer portion

Description

The **modf** function breaks down the floating-point value *x* into fractional and integer parts. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

Return Value

The **modf** function returns the signed fractional portion of *x*. There is no error return.

See Also

frexp, ldexp

Example

```
#include <math.h>
#include <stdio.h>


main()
 {
  double x,y,n;

  x = -14.87654321; /* Divide x into its fractional */
  y = modf(x,&n);   /* and integer parts            */

  printf("The modf(%f,&n) = %f and n = %f", x, y, n);
 }
```

This program uses **modf** to divide the floating-point value –14.87654321 into its fractional and integer parts.

Summary

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either **string.h** or **memory.h**

void movedata(*srcseg, srcoff, destseg, destoff, nbytes***);**

unsigned int *srcseg***;**	Segment address of source
unsigned int *srcoff***;**	Segment offset of source
unsigned int *destseg***;**	Segment address of destination
unsigned int *destoff***;**	Segment offset of destination
unsigned *nbytes***;**	Number of bytes

Description

The **movedata** function copies *nbytes* bytes from the source address specified by *srcseg:srcoff* to the destination address specified by *destseg:destoff*.

The **movedata** function is used to move far data in small- or medium-model programs where segment addresses of data are not implicitly known. In large-model programs, the **memcpy** or **memmove** function can be used since segment addresses are implicitly known.

Return Value

There is no return value.

See Also

FP_OFF, **FP_SEG**, **memcpy**, **memmove**, **segread**

Note

Segment values for the *srcseg* and *destseg* arguments can be obtained by using either the **segread** function or the **FP_SEG** macro.

The **movedata** function does not handle all cases of overlapping moves correctly (overlapping moves occur when part of the destination is the same memory area as part of the source). Overlapping moves are handled correctly in the **memmove** function.

Example

```
#include <memory.h>
#include <dos.h>
#include <malloc.h>

char far *src;
char far *dest;

main()
 {
  src  = _fmalloc(512);
  dest = _fmalloc(512);

  movedata(FP_SEG(src), FP_OFF(src), FP_SEG(dest),
           FP_OFF(dest), 512);
  printf("The data have been moved\n");
 }
```

This program uses **movedata** to move 512 bytes of data from `src` to `dest`.

_moveto

Summary

#include <graph.h>

struct xycoord {	
short xcoord;	x coordinate
short ycoord;	y coordinate
} far _moveto(*x*, *y*);	
short *x*, *y*;	Target position

Description

The **_moveto** function moves the current position to the logical point (*x*, *y*). No drawing takes place.

Return Value

The function returns the logical coordinates of the previous position as an **xycoord** structure, defined in **graph.h**.

See Also

_lineto

Example

```
#include <graph.h>
main()
{
 int loop, outloop;
 _setvideomode(_MRES16COLOR);
 for (outloop = 0; outloop < 20; outloop++) {
  for (loop = 0; loop < 320; loop += 7) {
   _setcolor( loop % 16 );
   _moveto( loop / 2, 0);
   _lineto(0, 199 - loop * 8 / 5 );
  }
  _selectpalette( outloop % 5 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program draws random line segments of different colors, calling **_moveto** to move between segments.

_msize, _fmsize, _nmsize

Summary

#include <malloc.h>	Required only for function declarations
size_t _msize(*buffer*);	Returns memory block size
**void ****buffer*;	Pointer to memory block
size_t _fmsize(*buffer*);	Returns far-heap memory block size
**void far ****buffer*;	Pointer to memory block
size_t _nmsize(*buffer*);	Returns near-heap memory block size
**void near ****buffer*;	Pointer to memory block

Description

The **_msize** function returns the size, in bytes, of the memory block allocated by a call to **calloc**, **malloc**, or **realloc**.

In large data models (compact- and large-model programs), **_msize** maps to **_fmsize**. In small data models (small- and medium-model programs), **_msize** maps to **_nmsize**.

The **_fmsize** function returns the size (in bytes) of the memory block allocated by a call to **_fmalloc**.

The **_nmsize** function returns the size (in bytes) of the memory block allocated by a call to **_nmalloc**.

Return Value

All three functions return the size (in bytes) as an unsigned integer.

See Also

calloc, **_expand**, **_fmalloc**, **malloc**, **_nmalloc**, **realloc**

Example

```
#include <stdio.h>
#include <malloc.h>

main()
 {
  long *oldbuffer;
  size_t newsize = 64000;
  oldbuffer = (long *)malloc(10000*sizeof(long));

  /* Get size of original memory: */
  printf("Size of memory block pointed to by oldbuffer = %u\n",
    _msize(oldbuffer));
  if (_expand(oldbuffer,newsize) != NULL)
  /* if _expand succeeded: */
    printf("Expand was able to increase block to %u\n",
      _msize(oldbuffer));
  /* otherwise _expand failed: */
  else
    printf("Expand was able to increase block to only %u\n",
      _msize(oldbuffer));
 }
```

Sample output:

```
Size of memory block pointed to by oldbuffer = 40000
Expand was able to increase block to only 44718
```

This program allocates a block of memory for `oldbuffer` and then uses **_msize** to display the size of that block. Next, it uses **expand** to expand the amount of memory used by `oldbuffer` and then calls **_msize** again to display the new amount of memory allocated to `oldbuffer`.

onexit

Summary

#include <stdlib.h> Required only for function declarations

onexit_t onexit(*func*); Pointer type **onexit_t** defined in **stdlib.h**
onexit_t *func*;

Description

The **onexit** function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to **onexit** create a register of functions that are executed "last-in, first-out." No more than 32 functions can be registered with **onexit**; **onexit** returns the value **NULL** if the number of functions exceeds 32. The functions passed to **onexit** cannot take parameters.

Note

The **onexit** function is not part of the ANSI definition but is instead a Microsoft extension. The ANSI-standard **atexit** function does the same thing **onexit** does and should be used instead of **onexit** where ANSI portability is desired.

Return Value

The **onexit** function returns a pointer to the function if successful, and returns **NULL** if there is no space left to store the function pointer.

See Also

exit

Example

```
#include <stdlib.h>

main( )
        {
        int fn1( ), fn2( ), fn3( ), fn4( );

        onexit(fn1);
        onexit(fn2);
        onexit(fn3);
        onexit(fn4);
        printf("This is executed first.\n");
        }

 int fn1( )
        {
        printf("next.\n");
        }

 int fn2( )
        {
        printf("executed ");
        }

 int fn3( )
        {
        printf("is ");
        }

 int fn4( )
        {
        printf("This ");
        }
```

Output:

```
This is executed first.
This is executed next.
```

open

Summary

#include <fcntl.h>	
#include <sys\types.h>	
#include <sys\stat.h>	
#include <io.h>	Required only for function declarations
int open(*path, oflag* ⟦, *pmode*⟧**);**	
char ****path*;**	File path name
int *oflag***;**	Type of operations allowed
int *pmode***;**	Permission setting

Description

The **open** function opens the file specified by *path* and prepares the file for subsequent reading or writing, as defined by *oflag*. The argument *oflag* is an integer expression formed by combining one or more of the following manifest constants, defined in **fcntl.h**. When more than one manifest constant is given, the constants are joined with the bitwise-OR operator (¦).

Constant	Meaning
O_APPEND	Repositions the file pointer to the end of the file before every write operation.
O_BINARY	Opens file in binary (untranslated) mode. (See **fopen** for a description of binary mode.)
O_CREAT	Creates and opens a new file for writing; this has no effect if the file specified by *path* exists.
O_EXCL	Returns an error value if the file specified by *path* exists. Only applies when used with **O_CREAT**.
O_RDONLY	Opens file for reading only; if this flag is given, neither **O_RDWR** nor **O_WRONLY** can be given.
O_RDWR	Opens file for both reading and writing; if this flag is given, neither **O_RDONLY** nor **O_WRONLY** can be given.

O_TEXT	Opens file in text (translated) mode. (See **fopen** for a description of text mode.)
O_TRUNC	Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed.
O_WRONLY	Opens file for writing only; if this flag is given, neither **O_RDONLY** nor **O_RDWR** can be given.

Note

Use the **O_TRUNC** flag with care, as it destroys the complete contents of an existing file.

Either **O_RDONLY**, **O_RDWR**, or **O_WRONLY** must be given to specify the access mode. The access mode does not default.

The *pmode* argument is required only when **O_CREAT** is specified. If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. The *pmode* is an integer expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in **sys\stat.h**. When both constants are given, they are joined with the bitwise-OR operator (¦). The meaning of the *pmode* argument is as follows:

Value	**Meaning**
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD ¦ S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read only. Under MS-DOS, all files are readable; it is not possible to give write-only permission. Thus the modes **S_IWRITE** and **S_IREAD ¦ S_IWRITE** are equivalent.

Important

Under MS-DOS Versions 3.0 and later with **SHARE** installed, a bug occurs when opening a new file with *oflag* set to **O_CREAT ¦ O_RDONLY** or **O_CREAT ¦ O_WRONLY** with *pmode* set to **S_IREAD**. In this case, the operating system will prematurely close the file during system calls made within **open**.

To get around the problem, open the file with the *pmode* argument set to **S_IWRITE**. After closing the file, call **chmod** and change the mode back to **S_IREAD**. Another work-around is to open the file with *pmode* set to **S_IREAD** and *omode* set to **O_CREAT ¦ O_RDWR**.

The **open** function applies the current file-permission mask to *pmode* before setting the permissions (see **umask**).

Return Value

The **open** function returns a file handle for the opened file. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Given path name is a directory; or an attempt was made to open a read-only file for writing; or a sharing violation occurred (the file's sharing mode does not allow the specified operations).
EEXIST	The **O_CREAT** and **O_EXCL** flags are specified, but the named file already exists.
EMFILE	No more file handles available (too many open files).
ENOENT	File or path name not found.

See Also

access, **chmod**, **close**, **creat**, **dup**, **dup2**, **fopen**, **sopen**, **umask**

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

main()
 {
  int fh1, fh2;

  fh1 = open("data1",O_RDONLY);
  if (fh1 == -1)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  fh2 = open("data2",O_WRONLY¦O_CREAT,S_IREAD¦S_IWRITE);
  if (fh2 == -1)
    perror("open failed on output file");
  else
    printf("open succeded on output file\n");
 }
```

This program uses **open** to open a file named `data1` for input and a file named `data2` for output.

outp, outpw

Summary

# include <conio.h>	Required only for function declarations
int outp(*port***,** *byte***);**	Outputs a byte
unsigned outpw(*port***,** *word***);**	Outputs a word
unsigned *port***;**	Port number
int *byte***;**	Output value
unsigned *word***;**	Output value

Description

The **outp** and **outpw** functions write a byte and a word, respectively, to the specified output port. The *port* argument can be any unsigned integer in the range 0 – 65,535; *byte* can be any integer in the range 0 – 255; and *word* can be any value in the range 0 – 65,535.

Return Value

The functions return the data output. There is no error return.

See Also

inp, **inpw**

Example

```
#include <conio.h>
#include <stdio.h>

int port, byte_val;

main()
 {
  port = 1;
  byte_val = 3;
  outp(port,byte_val);
  printf("The value %d has been output to port %d",
          byte_val, port);
 }
```

This program uses **outp** to write the value 3 to output port 1.

Summary

#include <graph.h>

void far _outtext(*text*)
char far **text*; Text to be output

Description

The **_outtext** function outputs the null-terminated string that *text* points to. No formatting is provided, in contrast to the standard console I/O library routines such as **printf**.

Text output begins at the current text position.

Return Value

There is no return value.

See Also

_setactivepage, **_settextposition**

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];
```

```
main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen: */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);     /* Turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on last line, it is out of window");
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_outtext** to print row and column coordinates at various points on the screen.

Summary

#include <stdio.h>	Required only for function declarations
void perror(*string*);	
const char **string*;	User-supplied message
int errno;	Error number
int sys_nerr;	Number of system messages
char *sys_errlist[sys_nerr];	Array of error messages

Description

The **perror** function prints an error message to **stderr**. The *string* argument is printed first, followed by a colon, the system error message for the last library call that produced the error, and a new-line character. If *string* is a null pointer or a pointer to a null string, **perror** prints only the system error message.

The actual error number is stored in the variable **errno**, which should be declared at the external level. The system error messages are accessed through the variable **sys_errlist**, which is an array of messages ordered by error number. The **perror** function prints the appropriate error message by using the **errno** value as an index to **sys_errlist**. The value of the variable **sys_nerr** is defined as the maximum number of elements in the **sys_errlist** array.

To produce accurate results, **perror** should be called immediately after a library routine returns with an error. Otherwise, the **errno** value may be overwritten by subsequent calls.

Return Value

The **perror** function returns no value.

See Also

clearerr, **ferror**, **strerror**

Note

> Under MS-DOS, some of the **errno** values listed in **errno.h** are not used. See Appendix A, "Error Messages," for a list of **errno** values used on MS-DOS and the corresponding error messages. The **perror** function prints an empty string for any **errno** value not used under MS-DOS.

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

main()
 {
  int fh1, fh2;

  fh1 = open("data1",O_RDONLY);
  if (fh1 == -1)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  fh2 = open("data2",O_WRONLY|O_CREAT,S_IREAD|S_IWRITE);
  if (fh2 == -1)
    perror("open failed on output file");
  else
    printf("open succeeded on output file\n");
 }
```

This program opens a file named `data1` for input and a file named `data2` for output. If either open operation fails, the program displays an error message to indicate the failure.

Summary

#include <graph.h>

short far _pie(*control, x1, y1, x2, y2, x3, y3, x4, y4*);	
short *x1, y1*;	Upper-left corner of bounding rectangle
short *x2, y2*;	Lower-right corner of bounding rectangle
short *x3, y3*;	Start vector
short *x4, y4*;	End vector

Description

The **_pie** function draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by the logical points (*x1, y1*) and (*x2, y2*). The arc starts where it interects the vector defined by (*x3, y3*) and ends where it interects the vector (*x4, y4*).

The wedge is drawn using the current color moving in a counterclockwise direction. The *control* parameter can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the figure using the current color and fill mask
_GBORDER	Does not fill the figure

Return Value

The function returns a nonzero value if the pie is drawn successfully; otherwise, it returns 0.

See Also

_arc, **_ellipse**, **_getcolor**, **_lineto**, **_rectangle**, **_setcolor**, **_setfillmask**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 _setvideomode(_MRES16COLOR);
 _pie(_GFILLINTERIOR, 80, 50, 240, 150, 240, 12, 0, 150 );
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws the “pie” wedge shown in Figure R.4.

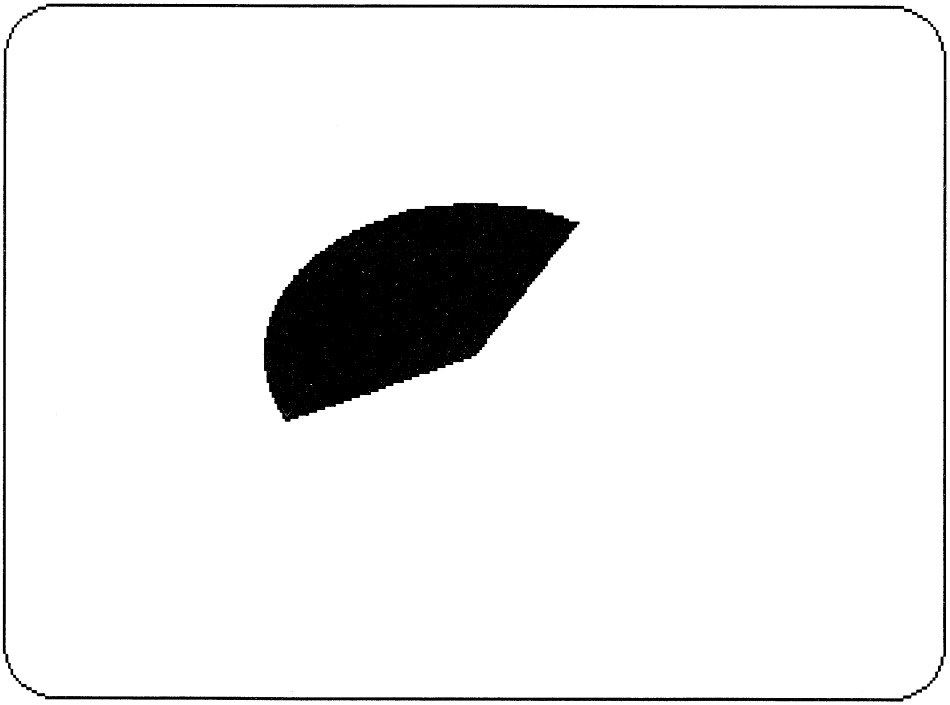


Figure R.4 Output of _pie Program

Summary

#include <math.h>

double pow(*x*, *y*);	
double *x*;	Number to be raised
double *y*;	Power of *x*

Description

The **pow** function computes *x* raised to the *y*th power.

Return Value

The **pow** function returns the value of x^y. If *x* is not 0 and *y* is 0, **pow** returns the value 1. If *x* is 0 and *y* is negative, **pow** sets **errno** to EDOM and returns HUGE_VAL. If both *x* and *y* are 0, or if *x* is negative and *y* is not an integer, the function prints a **DOMAIN** error message to **stderr**, sets **errno** to **EDOM**, and returns 0. If an overflow results, the function sets **errno** to **ERANGE** and returns ±HUGE_VAL. No message is printed on overflow or underflow.

The **pow** function does not recognize integral, floating-point values greater than 2^{64}, such as `1.0E100`.

See Also

exp, **log**, **sqrt**

Example

```
#include <math.h>
#include <stdio.h>

main()
 {
  double x = 2.0, y = 3.0, z;
  z = pow(x,y);                          /* z is 8.0 (2^3) */
  printf("The pow(%.2f,%.2f) = %.2f",x,y,z);
 }
```

This program uses **pow** to calculate the value of 2^3.

printf

Summary

#include <stdio.h>

int printf(*format*⟦**,** *argument*⟧**...);**	
**const char ****format*;	Format control

Description

The **printf** function formats and prints a series of characters and values to the standard output stream, **stdout**. The *format* consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. Ordinary characters and escape sequences are simply copied to **stdout** in order of their appearance. For example, the line

```
printf("Line one\n\t\tLine two\n");
```

produces the output

```
Line one
                Line two
```

(For more information on escape sequences, see Section 2.2.4, "Escape Sequences," in the *Microsoft C Language Reference.*)

If arguments follow the format, then the format must contain format specifications that determine the output format for these arguments. Format specifications, discussed below, always begin with a percent sign (%).

P-R

The format is read left to right. When the first format specification (if any) is encountered, the value of the first argument after the format is converted and output according to the format specification. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

A format specification has the following form:

%⟦*flags*⟧⟦*width*⟧⟦*.precision*⟧⟦{**F** | **N** | **h** | **l** | **L**}⟧*type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, or a number (see Table R.1). The simplest format specification contains only the percent sign and a type character (for example, `%s`). The optional fields control other aspects of the formatting, as follows:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><i>flags</i></td><td>Justification of output and printing of signs, blanks, decimal points, octal and hexadecimal prefixes (see Table R.2).</td></tr>
<tr><td><i>width</i></td><td>Minimum number of characters output.</td></tr>
<tr><td><i>precision</i></td><td>Maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values (see Table R.3).</td></tr>
<tr><td><b>F</b>, <b>N</b></td><td>Prefixes that allow user to override default addressing conventions of memory model being used, as shown below:
<table>
<tr><th>Prefix</th><th>Use</th></tr>
<tr><td><b>F</b></td><td>Used in small model to print value that has been declared <b>far</b></td></tr>
<tr><td><b>N</b></td><td>Used in medium, large, and huge models for <b>near</b> value</td></tr>
</table>
<p><b>F</b> and <b>N</b> should be used only with the <b>s</b> and <b>p</b> type characters, since they are relevant only to arguments that pass pointers.</p>
<p><b>F</b> and <b>N</b> are not part of the ANSI definition for <b>printf</b>, but are instead Microsoft extensions that should not be used if ANSI portability is desired.</p></td></tr>
<tr><td><b>h</b>, <b>l</b>, <b>L</b></td><td>Prefixes that determine size of argument expected, as shown below:
<table>
<tr><th>Prefix</th><th>Use</th></tr>
<tr><td><b>h</b></td><td>Used as a prefix with the integer types <b>d</b>, <b>i</b>, <b>o</b>, <b>x</b>, and <b>X</b> to specify that the argument is <b>short int</b>, or with <b>u</b> to specify a <b>short unsigned int</b></td></tr>
</table></td></tr>
</table>

l	Used as a prefix with **d**, **i**, **o**, **x**, and **X** types to specify that the argument is **long int**, or with **u** to specify a **long unsigned int**; also used as a prefix with **e**, **E**, **f**, **g**, and **G** types to specify a **double**, rather than a **float**
L	Used as a prefix with **e**, **E**, **f**, **g**, and **G** types to specify a **long double**

If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to **stdout**. For example, to print a percent-sign character, use %%.

Table R.1

Type Characters for printf

Character	Type	Output Format
d	**int**	Signed decimal integer
i	**int**	Signed decimal integer
u	**int**	Unsigned decimal integer
o	**int**	Unsigned octal integer
x	**int**	Unsigned hexadecimal integer, using "abcdef"
X	**int**	Unsigned hexadecimal integer, using "ABCDEF"
f	**double**	Signed value having the form [–]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
e	**double**	Signed value having the form [–]*d.dddd* **e** [*sign*]*ddd*, where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or –.
E	**double**	Identical to the **e** format, except that **E** introduces the exponent instead of **e**.

Table R.1 *(continued)*

Character	Type	Output Format
g	**double**	Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision (see below). The **e** format is used only when the exponent of the value is less than –4 or greater than the *precision* argument. Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
G	**double**	Identical to the **g** format, except that **G** introduces the exponent (where appropriate) instead of **e**.
c	**int**	Single character
s	String	Characters printed up to the first null character ('\ **0**') or until the *precision* value is reached.
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	Far pointer to **void**	Prints the address pointed to by the argument in the form *xxxx*:*yyyy*, where *xxxx* is the segment and *yyyy* is the offset, and the digits *x* and *y* are uppercase hexadecimal digits; **%Np** prints only the offset of the address, *yyyy*. Since **%p** expects a pointer to a far value, pointer arguments to **p** must be cast to **far** in small-model programs.

Table R.2

Flag Characters for printf

Flag[1]	Meaning	Default
–	Left justify the result within the given field width.	Right justify
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
blank (' ')	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank

Table R.2 *(continued)*

Flag[1]	Meaning	Default
#	When used with the **o**, **x**, or **X** format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank
	When used with the **e**, **E**, or **f** format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the **g** or **G** format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
	Ignored when used with **c**, **d**, **i**, **u**, or **s**	

---

[1] More than one flag can appear in a format specification.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or not-a-number, the **printf** function gives the following output:

Value	Output
+ infinity	**1.#INF***random-digits*
– infinity	**–1.#INF***random-digits*
Indefinite	*digit*.**#IND***random-digits*
Not-a-number	*digit*.**#NAN***random-digits*

The *width* argument is a non-negative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values (depending on whether the – flag is specified) until the minimum width is reached. If *width* is prefixed with a 0, zeros are added until the minimum width is reached (not useful for left-justified numbers).

The width specification never causes a value to be truncated; if the number of characters in the output value is greater than the specified width, or *width* is not given, all characters of the value are printed (subject to the precision specification).

The *width* specification may be an asterisk (*), in which case an **int** argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause a truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

The precision specification is a non-negative decimal integer preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits (see Table R.3). Unlike the width specification, the precision can cause truncation of the output value, or rounding in the case of a floating-point value.

The precision specification may be an asterisk (*), in which case an **int** argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the precision value, and the default when *precision* is omitted, depend on the type, as shown in Table R.3.

Table R.3

How printf Precision Values Affect Type

Type	Meaning	Default
d **i** **u** **o** **x** **X**	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	If *precision* is 0 or omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1.
e **E**	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if *precision* is 0 or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 0; if *precision* is explicitly 0, no decimal point appears.

Table R.3 *(continued)*

Type	Meaning	Default
g G	The precision specifies the maximum number of significant digits printed.	All significant digits are printed.
c	No effect	Character printed
s	The precision specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed.	Characters are printed until a null character is encountered.

■ Return Value

The **printf** function returns the number of characters printed.

■ See Also

fprintf, **scanf**, **sprintf**, **vfprintf**, **vprintf**, **vsprintf**

■ Example

```
main()          /* Format and print various data. */
   {
   char ch = 'h', *string = "computer";
   int count = 234, *ptr, hex = 0x10, oct = 010, dec = 10;
   double fp = 251.7366;

   printf("%d    %+d    %06d    %X    %x    %o\n\n",
        count, count, count, count, count, count);
   printf("1234567890123%n45678901234567890\n\n", &count);
   printf("Value of count should be 13; count = %d\n\n",
        count);
   printf("%10c%5c\n\n",ch,ch);
   printf("%25s\n%25.4s\n\n",string, string);
   printf("%f    %.2f    %e    %E\n\n",fp, fp, fp, fp);
   printf("%i    %i    %i\n\n", hex, oct, dec);
   ptr  = &count;
   printf("%Np    %p    %Fp\n",
        ptr, (int far *) ptr, (int far *) ptr);
   }
```

Output:

```
234    +234    000234    EA    ea    352

123456789012345678901234567890

Value of count should be 13; count = 13

         h    h

                 computer
                     comp

251.736600    251.74    2.517366e+002    2.517366E+002

16    8    10

127A    1328:127A    1328:127A
```

This program uses **printf** to display various strings, numbers, characters, and pointers with different formats.

putc, putchar

Summary

#include <stdio.h>

int putc(*c***,** *stream***);**	Writes a character to *stream*
int *c***;**	Character to be written
FILE **stream***;**	Pointer to **FILE** structure
int putchar(*c***);**	Writes a character to **stdout**
int *c***;**	Character to be written

Description

The **putc** routine writes the single character *c* to the output *stream* at the current position. The **putchar** routine is identical to **putc(***c***, stdout)**.

Return Value

The **putc** and **putchar** routines return the character written. A return value of **EOF** indicates an error, which could be caused by an attempt to write to a read-only file, specifying an invalid stream pointer, or no space left on the device. Since the **EOF** value is a legitimate integer value, the **ferror** function should be used to verify that an error occurred.

See Also

fputc, **fputchar**, **getc**, **getchar**

Note

The **putc** and **putchar** routines are identical to **fputc** and **fputchar**, respectively, but are macros, not functions.

Example

```
#include <stdio.h>

FILE *stream;
char buffer[81] = "This is the line of output\n";
int i, ch;

main()
 {
  stream = stdout;

  /* Write line to the stream: */
  for (i = 0; ( i < 81 ) &&
  ((ch = putc(buffer[i],stream)) != EOF); i++);

 }

/* Note that the body of the "for" statement is null, since the
** write operation is carried out in the test expression.
*/
```

This program uses **putc** to write `buffer` to a stream. If an error occurs, the program will stop before writing the entire buffer.

putch

Summary

#include <conio.h> Required only for function declarations

int putch(*c*)
int *c*; Character to be output

Description

The **putch** function writes the character *c* directly to the console.

Return Value

The function returns *c* if successful, and **EOF** if not.

C 4.0 Difference

In Microsoft C Version 4.0, **putch** returns an error code.

See Also

cprintf, **getch**, **getche**

Example

```
#include <conio.h>

mygetche()
 {
  int ch = getch();       /* Get a character with no echo */
  putch(ch);          /* Send the character to the console */
  return(ch);
 }

main()
 {
  char ch;
  while (( ch = mygetche()) != '\r');
 }
```

This program acts like **getche**, using **getch** and **putch**.

Summary

#include <stdlib.h>	Required only for function declarations
int putenv(*envstring*);	
**char ****envstring*;	Environment-string definition

Description

The **putenv** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program).

The *envstring* argument must be a pointer to a string with the form

varname=string

where *varname* is the name of the environment variable to be added or modified and *string* is the variable's value. If *varname* is already part of the environment, it is replaced by *string*; otherwise, the new *string* is added to the environment. A variable can be set to an empty value by specifying an empty *string*.

This function affects only the environment that is local to the currently running process; it cannot be used to enter new items in the command-level environment. When the currently running process terminates, the environment reverts to the level of the parent process (in most cases, the MS-DOS level). However, the environment affected by **putenv** is passed to any child processes created by **spawn** or **exec**, and these child processes set any new items added by **putenv**.

Never free a pointer to an environment entry because the environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to **putenv**, then exit the function in which the variable is declared.

Note

Environment-table entries must not be changed directly. If an entry must be changed, use **putenv**. To modify the returned value without affecting the environment table, use **strdup** or **strcpy** to make a copy of the string.

The **getenv** and **putenv** functions use the global variable **environ** to access the environment table. The **putenv** function may change the value of **environ**, thus invalidating the *envp* argument to the **main** function. Therefore, it's safer to use the **environ** variable to access the environment information.

Return Value

The **putenv** function returns 0 if it is successful. A return value of –1 indicates an error.

See Also

getenv

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>

main()
 {

   /* Attempt to change directory path: */
   if (putenv("PATH=a\bin;b:\tmp") == -1)
    {
     printf("putenv failed -- out of memory");
     exit(1);
    }
   else
    printf("'putenv' worked.\n");
}
```

This program uses **putenv** to change the value of the **PATH** variable in the environment table.

_putimage

Summary

#include <graph.h>

void far _putimage(*x, y, image, action*);	
short *x, y*;	Position of upper-left corner of image
char far **image*;	Stored image buffer
short *action*;	Interaction with existing screen image

Description

The **_putimage** function transfers to the screen the image stored in the buffer that *image* points to, placing the image's upper-left corner at the logical point (*x, y*). The *action* argument defines the interaction between the stored image and the one already on the screen. It may be any one of the following manifest constants (defined in **graph.h**):

Constant	Meaning
_GAND	Transfers the image over an existing image on the screen. The resulting image is the logical-AND product of the two images: points that had the same color in both the existing image and the new one will remain the same color, while points that have different colors are ANDed together.
_GOR	Superimposes the image onto an existing image. The new image does not erase the previous screen contents.
_GPRESET	Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by **_getimage**, producing a negative image.
_GPSET	Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by **_getimage**.
_GXOR	Causes the points on the screen to be inverted where a point exists in the *image* buffer. This behavior is exactly like that of the cursor: when an image is put against a complex background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The **_GXOR** constant is a special mode often used for animation.

Return Value

There is no return value.

See Also

_getimage

Example

```
#include <stdio.h>
#include <malloc.h>
#include <graph.h>

char far *buffer;

main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 for ( xvar = 163, loop = 0; xvar < 320; loop++, xvar += 3 ) {
  _setcolor(loop % 16 );
  yvar = xvar * 5 / 8;
  _rectangle(_GBORDER, 320-xvar, 200-yvar, xvar, yvar);
  _setcolor(rand(1) % 16 );
  _floodfill(0, 0, loop % 16 );
 }
 buffer = (char far *)malloc( (unsigned int)
          _imagesize( 0, 0, 80, 50 ) );
 if ( buffer == (char far *)NULL ) {
  exit( -1 );
 }
 _getimage(0, 0, 80, 50, buffer );
 _putimage( 80, 50, buffer, _GXOR );
 free((char *)buffer);
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a series of nested rectangles and stores a portion of the image in memory. It then calls **_putimage** to display it again.

puts

Summary

#include <stdio.h>

int puts(*string*);
const char ****string*;** String to be output

Description

The **puts** function writes *string* to the standard output stream **stdout**, replacing the string's terminating null character (**'\0'**) with a new-line character (**'\n'**) in the output stream.

Return Value

The **puts** function returns a 0 if successful. If the function fails, it returns a nonzero value.

C 4.0 Difference

In Microsoft C Version 4.0, **puts** returns the last character output or an EOF to indicate an error.

See Also

fputs, **gets**

Example

```
#include <stdio.h>

int result;

main()
 {
 /* Write a prompt to "stdout": */
   result = puts("Insert data disk and strike any key");
}
```

This program uses **putc** to write a string to **stdout**.

Summary

#include <stdio.h>

int putw(*binint, stream***);**	
int *binint***;**	Binary integer to be output
FILE **stream***;**	Pointer to **FILE** structure

Description

The **putw** function writes a binary value of type **int** to the current position of the *stream*. The **putw** function does not affect the alignment of items in the stream, nor does it assume any special alignment.

Return Value

The **putw** function returns the value written. A return value of **EOF** may indicate an error. Since **EOF** is also a legitimate integer value, **ferror** should be used to verify an error.

See Also

getw

Note

The **putw** function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with **putw**, since the size of an **int** and ordering of bytes within an **int** differ across systems.

Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

main()
 {
  stream = fopen("data.bin", "wb");

  if(putw(0345,stream) = EOF){      /* Write word to stream */
   if (ferror(stream)){             /* Make error check */
    printf("putw failed");
    clearerr(stream);
   }
   else
    ;
  }
  else
   printf( "\nputw wrote a word.\n" );
 }
```

This program uses **putw** to write a word to a stream and then performs an error check.

Summary

#include <stdlib.h>	For ANSI compatibility
#include <search.h>	Required only for function declarations

void qsort(*base, num, width,* **(***compare***)());**

void **base*;	Start of target array
size_t *num*;	Array size in elements
size_t *width*;	Element size in bytes
int (**compare*)(*elem1, elem2*);	*compare* function
const void **elem1, elem2*;	Array elements to compare

Description

The **qsort** function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. The **qsort** function overwrites this array with the sorted elements.

The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The **qsort** function calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values:

Value	Meaning
Less than 0	*element1* less than *element2*
0	*element1* equivalent to *element2*
Greater than 0	*element1* greater than *element2*

The sorted array is in increasing order, as defined by the *compare* function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the *compare* function.

Return Value

There is no return value.

See Also

bsearch, **lsearch**

Example

```
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare();                /* must declare as a function */
                              /* for qsort's compare        */

main (argc, argv)
int argc;
char **argv;
 {
  char **result;
  int i;

  /* Eliminate argv[0] from sort: */
  argv++;
  argc--;
  /* Sort remaining args using Quicksort algorithm: */
  qsort((void *)argv, (size_t)argc,sizeof(char *),compare);

  /* Output sorted list: */
  for (i=0;i<argc;++i)
    printf("%s\n", argv[i]);

 }

int compare (arg1, arg2)
char **arg1, **arg2;
 {

  /* Compare all of both strings: */
  return(strcmp(*arg1,*arg2));
 }
```

This program reads the command-line parameters and uses **qsort** to sort them. It then displays the sorted arguments.

Summary

#include <signal.h>

int raise(*sig*);	
int *sig*;	Signal to be raised

Description

The **raise** function sends *sig* to the executing program. The default action for *sig* will be taken unless a different action has been defined using the **signal** routine.

The signal can be one of the following manifest constants:

Signal	Meaning
SIGABRT	Abnormal termination. The default action terminates the calling program with exit code 3.
SIGFPE	Floating-point error. The default action terminates the calling program.
SIGILL	Illegal instruction. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action terminates the calling program.
SIGINT	CTRL+C interrupt. The default action issues INT 23H.
SIGSEGV	Illegal storage access. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action terminates the calling program.
SIGTERM	Termination request sent to the program. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action ignores it.

Return Value

If successful, the **raise** function returns 0. Otherwise, it returns a nonzero value.

See Also

abort, **signal**

Example

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

void handler();
float num   = 1.0;
float denom = 0.0;

main()
 {          /* Set so interrupt calls "handler" */
  if (signal(SIGFPE,handler) == (int(*)())-1)
   {
    perror("Couldn't set SIGFPE");
    abort();
   }

   if (denom == 0)
     raise(SIGFPE);
   else
     printf("Result of division is %f", num/denom);
 }

void handler()       /* Function called at Floating */
 {                   /* Point Exception interrupt     */
  char ch;

  printf("Inside Floating Point Exception Handler\n");
  exit(0);
 }
```

The above program uses **raise** to detect a division-by-zero error before the error is actually executed; it also sends control to the routine defined by **signal**.

Summary

#include <stdlib.h> Required only for function declarations

int rand(void);

Description

The **rand** function returns a pseudorandom integer in the range 0 – 32,767. The **srand** routine can be used to set a random starting point before calling **rand**.

Return Value

The **rand** function returns a pseudorandom number as described above. There is no error return.

See Also

srand

Example

```
#include <stdlib.h>
#include <stdio.h>

main()
 {
  int x;

  for (x = 1; x <= 20; x++)
    printf("Iteration %d, rand=%d\n", x, rand());
 }
```

This program displays the first 20 random integers generated by **rand**.

read

Summary

#include <io.h>	Required only for function declarations
int read(*handle, buffer, count***);**	
int *handle***;**	Handle referring to open file
char **buffer***;**	Storage location for data
unsigned int *count***;**	Maximum number of bytes

Description

The **read** function attempts to read *count* bytes into *buffer* from the file associated with *handle*. The read operation begins at the current position of the file pointer (if any) associated with the given file. After the read operation, the file pointer points to the next unread character.

Return Value

The **read** function returns the number of bytes actually read, which may be less than *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode (see below). The return value 0 indicates an attempt to read at end-of-file. The return value –1 indicates an error, and **errno** is set to the following value:

Value	Meaning
EBADF	The given *handle* is invalid; or the file is not open for reading; or the file is locked (MS-DOS Versions 3.0 and later only).

If you are reading more than 32K (the maximum size for type **int**) from a file, the return value should be of type **unsigned int**. (See the example that follows.) However, the maximum number of bytes that can be read from a file is 65,534 at a time, since 65,535 (or 0xFFFF) is indistinguishable from –1, and therefore would return an error.

If the file was opened in text mode, the return value may not correspond to the number of bytes actually read. When text mode is in effect, each carriage-return–line-feed pair is replaced with a single line-feed character. Only the single line-feed character is counted in the return value. The replacement does not affect the file pointer.

See Also

creat, **fread**, **open**, **write**

Note

> Under MS-DOS, when files are opened in text mode, a character is treated as an end-of-file indicator. When the CTRL+Z is encountered, the read terminates, and the next read returns 0 bytes. The file must be closed to clear the end-of-file indicator.

Example

```
#include <fcntl.h>  /* Needed only for O_RDWR definition */
#include <io.h>
#include <stdio.h>

char buffer[60000];

main()
 {
  int fh;
  unsigned int nbytes = 60000, bytesread;

  /* Open file for input: */
  if ((fh = open("data",O_RDONLY)) == -1)
   {
    perror("open failed on input file");
    exit(1);
   }

  /* Read in input: */
  if ((bytesread = read(fh,buffer,nbytes)) <= 0)
    perror("Problem reading file");
  else
    printf("Read %u bytes from file\n", bytesread);
 }
```

This program opens a file named `data` and tries to read 60,000 bytes from that file using **read**. It then displays the actual number of bytes read from `data`.

Summary

# include <stdlib.h>	For ANSI compatibility
# include <malloc.h>	Required only for function declarations
void *realloc(*buffer, size***);**	
void ****buffer*;**	Pointer to previously allocated memory block
size_t *size***;**	New size in bytes

Description

The **realloc** function changes the size of a previously allocated memory block. The *buffer* argument points to the beginning of the block. If *buffer* is **NULL**, **realloc** functions like **malloc** and allocates a new block of *size* bytes. If *buffer* is not **NULL**, it should be a pointer returned by **calloc**, **malloc**, or a prior call to **realloc**. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

The *buffer* argument can also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **malloc**, or **realloc** since the block was freed.

Return Value

The **realloc** function returns a **void** pointer to the reallocated memory block. The block may be moved when its size is changed; therefore, the *buffer* argument to **realloc** is not necessarily the same as the return value.

The return value is **NULL** if *size* is 0 or if there is insufficient memory available to expand the block to the given size. The original block is freed when this occurs.

C 4.0 Difference

Under Version 4.0 of Microsoft C, **malloc** allocates a zero-length item (that is, a header only) in the heap if *size* is 0. The resulting pointer can be passed to the **realloc** function to adjust the size at any time.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

See Also

calloc, **free**, **malloc**

Example

```
#include <malloc.h>
#include <stdio.h>

char *alloc;

main()
 {

  /* Get enough space for 50 characters: */
  alloc = malloc(50*sizeof(char));
  printf( "Block successfully allocated\n" );

  /* Reallocate block to hold 100 chars: */
  if (alloc != NULL)
    alloc = realloc(alloc,100*sizeof(char));

  if (alloc != NULL)
    printf("Block is successfully reallocated\n");
  else
    printf("'realloc' failed--block was freed\n");
}
```

This program allocates enough space for 50 characters, then uses **realloc** to expand this space so that it can hold 100 characters.

_rectangle

Summary

#include <graph.h>

short far _rectangle(*control*, *x1*, *y1*, *x2*, *y2***);**	
short *control*;	Fill flag
short *x1*, *y1*;	Upper-left corner
short *x2*, *y2*;	Lower-right corner

Description

The **_rectangle** function draws a rectangle with the current line style. The logical points (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle. The *control* parameter can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the figure with the current color using the current fill mask
_GBORDER	Does not fill the rectangle

If the current fill mask is **NULL**, no mask is used. Instead, the rectangle is filled with the current color.

Note

If you try to fill the rectangle with the **_floodfill** function, the rectangle must be bordered by a solid line-style pattern.

Return Value

The function returns a nonzero value if the rectangle is drawn successfully, or 0 otherwise.

See Also

_getcolor, **_getlinestyle**, **_setcolor**, **_setlinestyle**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 _setvideomode( _MRES16COLOR );
 _rectangle( _GBORDER, 80, 50, 240, 150 );
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws the rectangle shown in Figure R.5.

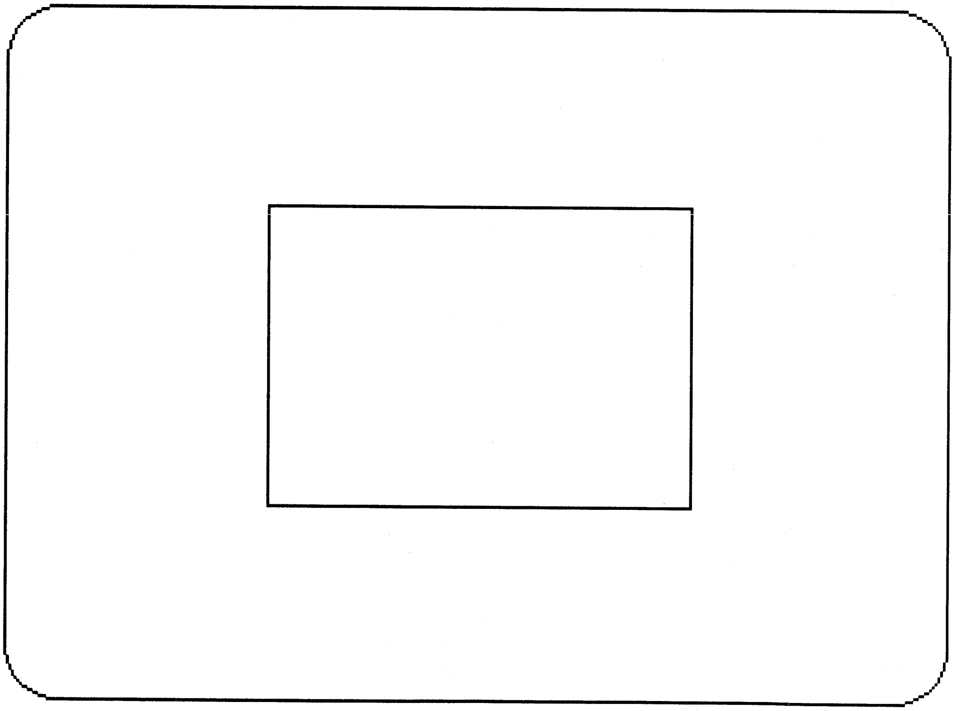


Figure R.5 Output of _rectangle Program

_remapallpalette, _remappalette

Summary

#include <graph.h>

short far _remapallpalette(*colors***):**
long far ****colors*;** Color number array

long far _remappalette(*pixel, color***);**
short *pixel***;** Pixel value to reassign
long *color***;** Color number to assign pixel to

Description

The **_remapallpalette** function remaps all of the available pixel values simultaneously. The *colors* argument points to an array of color numbers. The default array of color numbers is shown below:

No.	Color	No.	Color
0	Black	8	Dark gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	Bright white

The number of colors mapped depends on the number of colors supported by the current video mode. A 16-color mode will map all the colors shown, from 0 to 15. An eight-color mode will map only the colors from 0 to 7. A four-color mode will use one of the standard palettes supported by the EGA (see **_selectpalette**). A two-color mode will support the colors 0 and 7 (black and white).

For example, if *colors* is an eight-element array whose members are 1, 3, 5, 7, 2, 4, 6, and 8, these pixel values would be mapped to actual colors in an eight-color video mode as shown below:

Pixel No.	Color	Pixel No.	Color
1	Blue	5	Magenta
2	Green	6	Brown
3	Cyan	7	White
4	Red	8	Dark gray

In general, you are simply mapping a set of pixel value numbers to the ordinal numbers recognized by the hardware. The *colors* array can be larger than the number of colors supported by the current video mode; only the first *n* colors are available, where *n* is the number of colors supported by the current video mode.

The **_remappalette** function remaps one pixel value to *color*, which must be a color number supported by the current video mode.

Both functions immediately affect the current display.

The VGA color graphics modes support a palette of 262,144 colors (equivalent to 256K). Three bytes of data represent the intensities of red, blue, and green. In each byte, the two high-order bits must be 0. The remaining six bits represent the intensity of blue, green, and red (reading from high-order byte to low-order byte). For example, to make a low-intensity white color, equal values of red, green, and blue are used. Thus, the three-byte color number would be:

```
blue byte              green byte           red byte
00011111               00011111             00011111
high -----------------------------------> low order
```

Because of the splitting of colors between bytes, the color numbers are not continuous. Manifest constants for the default color numbers are supplied to provide compatibility with EGA practice. The names of these constants are listed below:

Number	Constant	Number	Constant
0	_BLACK	8	_GRAY
1	_BLUE	9	_LIGHTBLUE
2	_GREEN	10	_LIGHTGREEN
3	_CYAN	11	_LIGHTCYAN
4	_RED	12	_LIGHTRED
5	_MAGENTA	13	_LIGHTMAGENTA
6	_BROWN	14	_LIGHTYELLOW
7	_WHITE	15	_BRIGHTWHITE

Note

Both the **_remappalette** and **_remapallpalette** functions work in all video modes, but only with an EGA or VGA.

Return Value

If successful, **_remapallpalette** returns 0 and **_remappalette** returns the previous color number of the *pixel* argument. If either function is inoperative (that is, running under a configuration other than an EGA or VGA), it returns –1.

See Also

_selectpalette, **_setvideomode**

Example

```
#include <stdio.h>
#include <graph.h>

int colorsarray[ 32 ];
```

```
main()
{
 int xvar, yvar, loop = 0;
 int loop1;
 _setvideomode(_MRES16COLOR);
 /* make 16 rectangles */
 for ( loop = 0; loop < 32; loop += 2) {
  _setcolor( loop % 16 );
  _rectangle(_GFILLINTERIOR,loop*10,95,(loop+1)*10,105);
 }
 /* Do this loop until a character is typed */
 while ( !kbhit() ) {
  for ( loop = 0; loop < 100; loop++ ) {
   colorsarray[ loop % 32 ] = rand(1) % 16;
  } /* change all colors eventually */
  _remapallpalette((char *)(&(colorsarray[0])));
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a series of different-colored overlapping rectangles, while continuously changing the palette to random values.

remove

Summary

#include <io.h>	Required only for function declarations
#include <stdio.h>	Use either **io.h** or **stdio.h**
int remove(*path*);	
const char **path*;	Path name of file to be removed

Description

The **remove** function deletes the file specified by *path*.

Return Value

The function returns 0 if the file is successfully deleted. Otherwise, it returns –1 and sets **errno** to one of these values:

Value	Meaning
EACCES	Path name specifies a directory or a read-only file
ENOENT	File or path name not found

See Also

close, **unlink**

Example

```
#include <io.h>
#include <stdio.h>

main()
{
 int result = remove("data");
 if (result == -1)
  perror("Could not delete 'data'");
 else
  printf("'data' was successfully deleted\n");
}
```

This program uses **remove** to delete a file named `data`.

Summary

#include <io.h>	Required only for function declarations
#include <stdio.h>	Use either **io.h** or **stdio.h**
int rename(*oldname***,** *newname***);**	
const char **oldname*;	Pointer to old name
const char **newname*;	Pointer to new name

Description

The **rename** function renames the file or directory specified by *oldname* to the name given by *newname.* The old name must be the path name of an existing file or directory. The new name must not be the name of an existing file or directory.

The **rename** function can be used to move a file from one directory to another by giving a different path name in the *newname* argument. However, files cannot be moved from one device to another (for example, from drive A to drive B). Directories can only be renamed, not moved.

Return Value

The **rename** function returns 0 if it is successful. On an error, it returns a nonzero value and sets **errno** to one of the following values:

Value	Meaning
EACCES	File or directory specified by *newname* already exists or could not be created (invalid path); or *oldname* is a directory and *newname* specifies a different path
ENOENT	File or path name specified by *oldname* not found
EXDEV	Attempt to move a file to a different device

See Also

creat, **fopen**, **open**

Note

Note that the order of arguments in **rename** is the reverse of their order in versions of the Microsoft C Compiler prior to 4.0. This change was made to conform to the proposed ANSI C standard.

■ Example

```
#include <io.h>
#include <stdio.h>

main()
 {
  int result;

  /* Attempt to rename file: */
  result = rename("input","data");

  /* Check for errors: */
  if (result != 0)
    perror("Was not able to rename file");
  else
    printf("File successfully renamed");
 }
```

This program uses **rename** to rename a file named `input` to `data`. For this operation to succeed, a file named `input` must exist and a file named `data` must not exist.

Summary

#include <stdio.h>

void rewind(*stream*);
FILE *_stream_; Pointer to **FILE** structure

Description

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. A call to **rewind** is equivalent to

(void) fseek(*stream*, 0L, SEEK_SET);

except that **rewind** clears the end-of-file and error indicators for the stream, and **fseek** does not; also, **fseek** returns a value that indicates whether the pointer was successfully moved, but **rewind** does not return any value.

See Also

fseek, **ftell**

Example

```
#include <stdio.h>

FILE *stream;
int data1, data2;

main()
 {
  data1 = 1;
  data2 = -37;

  stream = fopen("data","w+");
  fprintf(stream,"%d %d",data1,data2);
  rewind(stream);
  fscanf(stream,"%d",&data1);
  fscanf(stream,"%d",&data2);
  printf("The values read back in are: %d and %d\n",
          data1,data2);
 }
```

This program first opens a file named `data` for input and output and writes two integers to the file. Next, it uses **rewind** to reposition the file pointer to the beginning of the file and reads the data back in.

Summary

#include <direct.h>	Required only for function declarations
int rmdir(*path*);	
**char ****path*;	Path name of directory to be removed

Description

The **rmdir** function deletes the directory specified by *path*. The directory must be empty, and it must not be the current working directory or the root directory.

Return Value

The **rmdir** function returns the value 0 if the directory is successfully deleted. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	The given path name is not a directory; or the directory is not empty; or the directory is the current working directory or root directory.
ENOENT	Path name not found.

See Also

chdir, **mkdir**

Example

```
#include <direct.h>
#include <stdio.h>

main()
 {
  int result1, result2;

  /* Remove "/sub1" from root directory: */
  result1 = rmdir("/sub1");
  if (result1 == -1)
    perror("Unable to remove directory");
  else
    printf("Directory successfully removed");

  /* Remove subdirectory "sub2" from */
  /*  the current working directory:  */
  result2 = rmdir("sub2");
  if (result2 == -1)
    perror("Unable to remove directory");
  else
    printf("Directory successfully removed");
 }
```

This program uses **rmdir** to remove the subdirectory `\sub1` from the root directory and the subdirectory `sub2` from the current working directory.

Summary

#include <stdio.h>

int rmtmp(void);

Description

The **rmtmp** function is used to clean up all the temporary files in the current directory. The function removes only those files created by **tmpfile** and should be used only in the same directory in which the temporary files were created.

Return Value

The **rmtmp** function returns the number of temporary files closed and deleted.

See Also

flushall, **tmpfile**, **tmpnam**

Example

```
#include <stdio.h>
FILE *stream;

main()
 {
  int numdeleted;

  /* Create a temporary file: */
  if ((stream = tmpfile()) == NULL)
    perror("Could not open new temporary file");
  /* Remove a temporary file: */
  numdeleted = rmtmp();
  printf("Number of closed files deleted in current "
         "directory = %d\n", numdeleted);
 }
```

This program creates a temporary file, then uses **rmtmp** to delete this file.

_rotl, _rotr

Summary

#include <stdlib.h>

unsigned int _rotl(*value*, *shift*);	Rotate left
unsigned int _rotr(*value*, *shift*);	Rotate right
unsigned int *value*;	Value to be rotated
int *shift*;	Number of bits to shift

Description

The **_rotl** and **_rotr** functions rotate *value* by *shift* bits.

Return Value

Both functions return the rotated value. There is no error return.

See Also

_lrotl, _lrotr

Example

```
#include <stdlib.h>

main()
{
 unsigned int val = 0x0123;
 printf("_rotl(val,2)   = 0x%4.4x\n", _rotl(val,2));
 printf("_rotr(val,8)   = 0x%4.4x\n", _rotr(val,8));
}
```

The output would look like this:

```
_rotl(val,2)   = 0x048c
_rotr(val,8)   = 0x2301
```

This program uses **_rotr** and **_rotl** with different *shift* values to rotate the integer value `0x123`.

Summary

# include <malloc.h>	Required only for function declarations
void *sbrk(*incr*);	
int *incr*;	Number of bytes added or subtracted

Description

The **sbrk** function resets the break value for the calling process. The break value is the address of the first byte of memory beyond the end of the default data segment. The **sbrk** function adds *incr* bytes to the break value; the size of the process' allocated memory is adjusted accordingly. Note that *incr* may be negative, in which case the amount of allocated space is decreased by *incr* bytes.

Return Value

The **sbrk** function returns the previous break value. A return value of (**char ***)–1 indicates an error, and **errno** is set to **ENOMEM**, indicating that insufficient memory was available.

See Also

calloc, **free**, **malloc**, **realloc**

Important

> In compact-, large-, and huge-model programs, **sbrk** fails and returns (**char ***)–1. Use **malloc** for allocation requests in large-model programs.

Example

```
#include <malloc.h>
#include <stdio.h>

main()
 {
  void *alloc;

  alloc = sbrk(100);      /* 100 bytes allocated on the heap */

  if (alloc != (char *)-1)
   {
    printf("100 bytes of memory have been allocated\n");
    printf("Now 40 bytes will be deallocated\n");

    sbrk(-40);          /* Deallocate 40 bytes */
   }
  else
    perror("Problem allocating 100 bytes\n");
 }
```

This program uses **sbrk** to allocate 100 bytes of memory and then to deallocate 40 bytes.

Summary

#include <stdio.h>

int scanf(*format*⟦, *argument*⟧...);
**const char ****format*; Format control

Description

The **scanf** function reads data from the standard input stream **stdin** into the locations given by *argument*. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields. The format can contain one or more of the following:

- White-space characters ; blank (**' '**); tab (**'\t'**); or new line (**'\n'**). A white-space character causes **scanf** to read, but not store, all consecutive white-space characters in the input up to the next non-white-space character. One white-space character in the format matches any number (including 0) and combination of white-space characters in the input.
- Non-white-space characters, except for the percent sign (%). A non-white-space character causes **scanf** to read, but not store, a matching non-white-space character. If the next character in **stdin** does not match, **scanf** terminates.
- Format specifications, introduced by the percent sign (%). A format specification causes **scanf** to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The format is read from left to right. Characters outside format specifications are expected to match the sequence of characters in **stdin**; the matched characters in **stdin** are scanned but not stored. If a character in **stdin** conflicts with the format, **scanf** terminates. The conflicting character is left in **stdin** as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to the format specification and stored in the location specified by the first *argument*. The second format specification causes the second input field to be converted and stored in the second *argument*, and so on through the end of the format string.

An input field is defined as all characters up to the first white-space character (space, tab, or new line), or up to the first character that cannot be converted according to the format specification, or until the field width, if specified, is reached, whichever comes first. If there are too many arguments for the given format specifications, the extra arguments are evaluated but ignored. The results are undefined if there are not enough arguments for the given format specifications.

A format specification has the following form:

%⟦*⟧⟦*width*⟧⟦{**F** | **N**}⟧⟦{**h** | **l**}⟧*type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters — that is, a sequence of characters that must match the input. For example, to specify that a percent-sign character is to be input, use `%%`.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The *width* is a positive decimal integer controlling the maximum number of characters to be read from **stdin**. No more than *width* characters are converted and stored at the corresponding *argument*. Fewer than *width* characters may be read if a white-space character (space, tab, or new line) or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional **F** and **N** prefixes allow the user to override the default addressing conventions of the memory model being used. **F** should be prefixed to an *argument* pointing to a **far** object, while **N** should be prefixed to an *argument* pointing to a **near** object. Note also that the **F** and **N** prefixes are not part of the ANSI definition for **scanf**, but are instead Microsoft extensions which should not be used when ANSI portability is desired.

The optional prefix **l** indicates that the **long** version of the following type is to be used, while the prefix **h** indicates that the **short** version is to be used. The corresponding *argument* should point to a **long** or **double** object (with the **l** character) or a **short** object (with the **h** character). The **l** and **h** modifiers can be used with the **d**, **i**, **n**, **o**, **x**, and **u** type characters. The **l** modifier can also be used with the **e**, **f**, and **g** type characters. The **l** and **h** modifiers are ignored if specified for any other type.

The type characters and their meanings are described in Table R.4.

Table R.4

Type Characters for scanf

Character	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to **int**
D	Decimal integer	Pointer to **long**
o	Octal integer	Pointer to **int**
O	Octal integer	Pointer to **long**
x	Hexadecimal integer[1]	Pointer to **int**
X	Hexadecimal integer[1,2]	Pointer to **long**
i	Decimal, hexadecimal, or octal integer	Pointer to **int**
I	Decimal, hexadecimal, or octal integer	Pointer to **long**
u	Unsigned decimal integer	Pointer to **unsigned int**
U	Unsigned decimal integer	Pointer to **unsigned long**
e, **E** **f** **g**, **G**	Floating-point value consisting of an optional sign (+ or –), a series of one or more decimal digits possibly containing a decimal point, and an optional exponent ("e" or "E") followed by an optionally signed integer value	Pointer to **float**

Table R.4 *(continued)*

Character	Type of Input Expected	Type of Argument
c	Character. White-space characters that are ordinarily skipped are read when **c** is specified; to read the next non-white-space character, use `%1s`.	Pointer to **char**
s	String	Pointer to character array large enough for input field plus a terminating null character (**'\0'**), which is automatically appended
n	No input read from stream or buffer	Pointer to **int**, into which is stored the number of characters successfully read from the stream or buffer up to that point in the current call to **scanf**
p	Value in the form *xxxx:yyyy*, where the digits *x* and *y* are uppercase hexadecimal digits	Pointer to far pointer to **void**

---

[1] Since the input for a %x or %X format specifier is always interpreted as a hexadecimal number, the input should not include a leading **0x** or **0X**. (If **0x** or **0X** is included, the **0** is interpreted as a hexadecimal input value.)

[2] The X type is not part of the ANSI definition for **scanf** but is instead a Microsoft extension and should not be used when ANSI portability is desired.

To read strings not delimited by space characters, a set of characters in brackets ([ ]) can be substituted for the **s** (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that *does* appear in the rest of the character set.

To store a string without storing a terminating null character (**'\0'**), use the specification %*n***c,** where *n* is a decimal integer. In this case, the **c** type character indicates that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no null character (**'\0'**) is appended. If **n** is not specified, the default value for it is 1.

The **scanf** function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons: the specified width has been reached; the next character cannot be converted as specified; the next character conflicts with a character in the control string that it is supposed to match; or the next character fails to appear in a given character set. For whatever reason, when **scanf** stops reading an input field, the next input field is considered to begin at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on **stdin**.

■ Return Value

The **scanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

■ See Also

fscanf, **printf**, **sscanf**, **vfprintf**, **vprintf**, **vsprintf**

■ Examples

```
#include <stdio.h>

FILE *stream;
int i;
float fp;
char c, s[81];

main()
 {
  int result;

  printf("Enter an integer, a floating point number, \n
  a character and a string\n>> ");
  result = scanf("%d %f %c %s", &i, &fp, &c, s);

  printf("\nThe number of fields input is %d\n", result);
  printf("The contents are: %d %f %c %s\n", i, fp, c, s);
 }
```

The example above uses **scanf** to read various types of data from **stdin**.

```
#include <stdio.h>

main()
 {
  int numassigned, val;

  printf("Enter hexadecimal or octal #, or 00 to quit:\n");
  do
   {
    printf("# = ");
    /* Input octal or hex value: */
    numassigned = scanf("%i",&val);
    printf("Decimal # = %i\n",val);
   }
  while (val && numassigned);
/* Loop ends if input value is 00 or  */
/*  "scanf" is unable to assign field */
 }
```

Sample output:

```
Enter hexadecimal or octal #, or 00 to quit:
# = 0xf
Decimal # = 15
# = 0100
Decimal # = 64
# = 00
Decimal # = 0
```

The above example uses **scanf** to read hexadecimal and octal values. It uses **printf** to convert these values to decimal and display them.

Summary

#include <stdlib.d>

void _searchenv(*name, env_var, path***);**

char **name*;	Name of file to search for
char **env_var*;	Environment to search
char **path*;	Buffer to store complete path

Description

The **_searchenv** routine searches for the target file in the specified domain. The *env_var* variable can be any environment variable which specifies a list of directory paths, such as **PATH**, **LIB**, **INCLUDE**, or other user-defined variables. It will most often be **PATH**, which searches for *fname* on all paths specified in the **PATH** variable.

The routine first searches for the file in the current working directory. If it doesn't find the file, it next looks through the directories specified by the environment variable.

If the target file is found in one of the directories, the newly created path is copied into the buffer that *path* points to. You must ensure that there is sufficient space for the constructed path name. If the *target* file is not found, *path* will contain an empty null-terminated string.

Example

```
#include <dos.h>

main()
 {
  char path_buffer [40];

  /* search for file at root level */
  _searchenv ("autoexec.bat", "PATH", path_buffer);
  printf ("path: %s\n", path_buffer);

  /* search for file in subdirectory */
  _searchenv ("searchen.c", "PATH", path_buffer);
  printf ("path: %s\n", path_buffer);

 }
```

This program searches for the specified files.

segread

Summary

# include <dos.h>

void segread(*segregs*);	
**struct SREGS ****segregs*;	Segment-register values

Description

The **segread** function fills the structure pointed to by *segregs* with the current contents of the segment registers. The **SREGS** union is described in the reference page for **int86x**. This function is intended to be used with the **intdosx** and **int86x** functions to retrieve segment-register values for later use.

Return Value

There is no return value.

See Also

FP_SEG, **intdosx**, **int86x**,

Example

```
#include <dos.h>
struct SREGS segregs;
unsigned int cs, ds, es, ss;

main()
 {
  segread(&segregs); /* Read the segment register values */
  cs = segregs.cs;
  ds = segregs.ds;
  es = segregs.es;
  ss = segregs.ss;
  printf("cs = %x, ds = %x, es = %x, ss = %x\n", cs, ds, es, ss);
 }
```

This program uses **segread** to obtain the current values of the segment registers, then displays these values.

Summary

#include <graph.h>

short far _selectpalette(*number*);
short *number*; Palette number

Description

The **_selectpalette** function works only under the **MRES4COLOR** and **MRESNOCOLOR** video modes. A palette consists of a selectable background color (Color 0) and three set colors. Under the **MRES4COLOR** mode the *number* argument selects one of the four predefined palettes shown in Table R.5.

Table R.5

MRES4COLOR Palette Colors

Palette Number	Pixel Values		
	Color 1	**Color 2**	**Color 3**
0	Green	Red	Brown
1	Cyan	Magenta	Light gray
2	Light green	Light red	Yellow
3	Light cyan	Light magenta	White

The **MRESNOCOLOR** video mode is used with black-and-white displays, producing palettes consisting of various shades of grey. It will also produce color when used with a color display. The number of palettes available depends upon whether a CGA or EGA hardware package is employed. Under a CGA configuration, only the two palettes shown in Table R.6 are available.

Table R.6

MRESNOCOLOR Mode
CGA Palette Colors

Palette Number	Pixel Values		
	Color 1	Color 2	Color 3
0	Blue	Red	Light gray
1	Light blue	Light red	White

Under the EGA configuration, the three palettes shown in Table R.7 are available in the **MRESNOCOLOR** video mode.

Table R.7

MRESNOCOLOR Mode
EGA Palette Colors

Palette Number	Pixel Values		
	Color 1	Color 2	Color 3
0	Green	Red	Brown
1	Light green	Light red	Yellow
2	Light cyan	Light red	Yellow

Note

> With an EGA in **MRESNOCOLOR** video mode, Palette 3 is identical to Palette 1.

Return Value

The function returns the value of the previous palette. There is no error return.

See Also

_getvideoconfig, **_setvideomode**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop, outloop;
 _setvideomode(_MRES4COLOR);
 for (outloop = 0; outloop < 20; outloop++) {
  for (loop = 0; loop < 320; loop += 7) {
   _setcolor( loop % 16 );
   _moveto( loop / 2, 0);
   _lineto(0, 199 - loop * 8 / 5 );
  }
  _selectpalette( outloop % 5 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a series of line segments while continuously changing the current palette.

_setactivepage

Summary

#include <graph.h>

short far _setactivepage(*page*);
short *page*;		Memory page number

Description

For hardware/mode configurations with enough memory to support multiple-screen pages, **_setactivepage** specifies the area in memory where graphics output is written. The *page* argument selects the current active page. The default page number is 0.

Screen animation is done by alternating the graphics pages displayed. Use the **_setvisualpage** function to display a completed graphics page while executing graphics statements in another active page.

These functions can also be used to control text output if you use the text functions **_outtext**, **_settextposition**, **_gettextposition**, **_settextcolor**, **_gettextcolor**, **_settextwindow**, and **_wrapon** instead of the standard C-language I/O functions. See Section 4.7.7, "Output Text."

Note

> The CGA hardware configuration has only 16K of RAM available to support multiple video pages, and only in the text mode. The EGA and VGA configurations may be equipped with up to 256K of RAM for multiple video pages in graphics mode.

Return Value

If successful, the function returns the page number of the previous active page. If the function fails, it returns a negative value.

See Also

_gettextcolor, **_gettextposition**, **_outtext**, **_settextcolor**, **_settextposition**, **_settextwindow**, **_setvideomode**, **_setvisualpage**, **_wrapon**

■ Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop = 0;
 _setvideomode( _MRES16COLOR );
 /* Repeat until a character is typed */
 while ( !kbhit() ) {
  /* alternate between page 0 and page 1 */
  _setactivepage( loop & 1 );
  _setcolor( loop % 16 );
  _rectangle( _GFILLINTERIOR, 80, 50, 240, 150 );
  _setvisualpage( loop++ & 1 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program creates an animated rectangle whose color changes between screen alternations.

_setbkcolor

■ Summary

#include <graph.h>

long far _setbkcolor(*color***);**
long *color***;** Desired color value

■ Description

The **_setbkcolor** function sets the current background color to the pixel value *color*.

Since the background color is Color 0, the **_remappalette** function will do the same thing that **_setbkcolor** does. Unlike **_remappalette**, however, **_setbkcolor** does not require an EGA or VGA environment. The **_setbkcolor** function does not affect anything already appearing on the display (only the subsequent output), while in a graphics mode it immediately changes all background pixels.

■ Return Value

This function returns the pixel value of the old background color. There is no error return.

■ See Also

_getbkcolor, **_remappalette**, **_selectpalette**

S

■ Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop;
 _setvideomode(_MRES16COLOR);
 for (loop = 0; loop < 20; loop++ ) {
  /* Get the next background color */
  _setbkcolor((_getbkcolor() + 1 ) % 16 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program steps through 20 different background colors.

setbuf

Summary

#include <stdio.h>

void setbuf(*stream, buffer***);**	
FILE ****stream*;**	Pointer to **FILE** structure
char ****buffer*;**	User-allocated buffer

Description

The **setbuf** function allows the user to control buffering for *stream*. The argument *stream* must refer to an open file before it has been read or written. If the *buffer* argument is **NULL**, the stream is unbuffered. If not, the buffer must point to a character array of length **BUFSIZ**, where **BUFSIZ** is the buffer size as defined in **stdio.h**. The user-specified buffer is used for I/O buffering instead of the default system-allocated buffer for the given stream.

The **stderr** and **stdaux** streams are unbuffered by default but may be assigned buffers with **setbuf**.

Return Value

There is no return value.

See Also

fclose, **fflush**, **fopen**

Example

```
#include <stdio.h>

char buf[BUFSIZ];
FILE *stream1, *stream2;

main()
 {
  stream1 = fopen("data1","r");
  stream2 = fopen("data2","w");

  /* "stream1" uses user-assigned buffer: */
  setbuf(stream1,buf);

  /* "stream2" is unbuffered             */
  setbuf(stream2,NULL);
  printf("Buffering of the streams has been set");
 }
```

This program first opens files named `data1` and `data2`. Then it uses **setbuf** to give `data1` a user-assigned buffer and to change `data2` so that it has no buffer.

_setcliprgn

■ Summary

#include <graph.h>

void far _setcliprgn(*x1, y1, x2, x2*);	
short *x1, y1*;	Upper-left corner of clip region
short *x2, y2*;	Lower-right corner of clip region

■ Description

The **_setcliprgn** function limits the display of subsequent graphics calls to the part that fits within a particular area of the screen, known as the "clipping region." The physical points (*x1, y1*) and (*x2, y2*) are the diagonally opposed sides of a rectangle that defines the clipping region. This function does not change the logical-coordinate system. Rather, it merely masks the screen.

Note

> The **_setcliprgn** function affects graphics output only. To mask the screen for text output, use the **_settextwindow** function.

■ Return Value

There is no return value.

■ See Also

_setviewport, **_settextwindow**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 _setvideomode(_MRES16COLOR);
 _setcliprgn( 0, 0, 200, 125 );
 _ellipse ( _GFILLINTERIOR, 80, 50, 240, 200 );
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws an ellipse lying partly within a clipping region, as shown in Figure R.6.

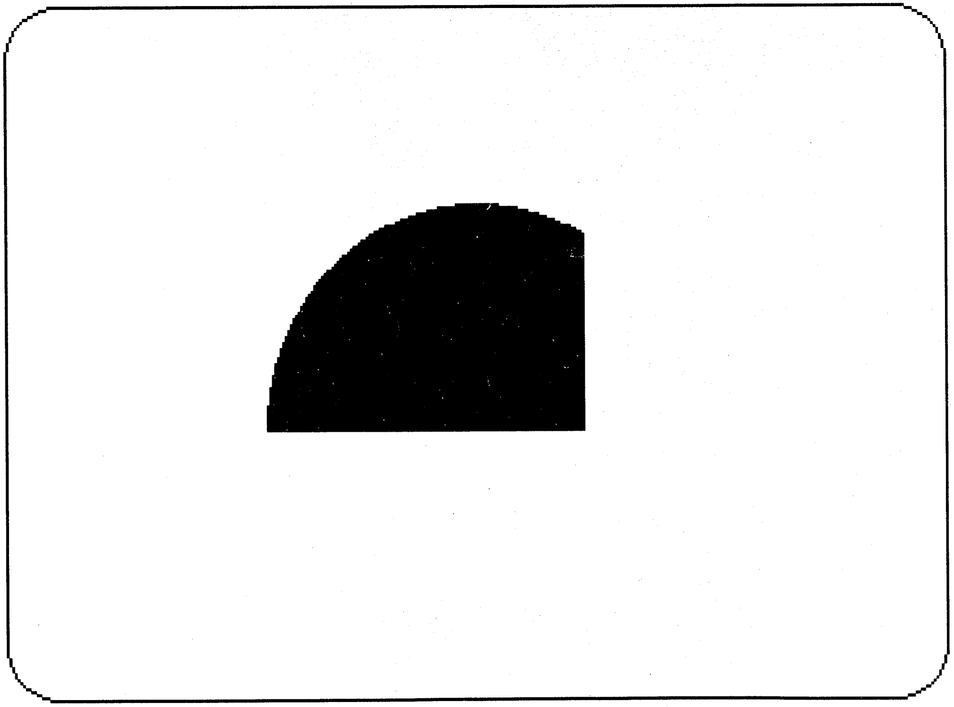


Figure R.6 Output of _setcliprgn Program

_setcolor

Summary

#include <graph.h>

short far _setcolor(*color***);**
short *color*; Desired color number

Description

The **_setcolor** function sets the current color parameter to *color*. The *color* parameter is masked so as to always be within range. All of the drawing functions use the current color (**_arc**, **_ellipse**, **_floodfill**, **_lineto**, **_pie**, **_rectangle**, and **_setpixel**).

The default color is the highest numbered color in the current palette.

Return Value

There is no return value.

See Also

_arc, **_ellipse**, **_floodfill**, **_getcolor**, **_lineto**, **_pie**, **_rectangle**, **_selectpalette**, **_setpixel**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop, loop1;
 _setvideomode(_MRES16COLOR);
 for (loop1 = 0; loop1 < 20; loop1++) { /* Get next color: */
  _setcolor((_getcolor() + 1) % 16 );
  for (loop = 0; loop < 3200; loop++) {
   /* Set a random pixel normalized to be on the screen */
   _setpixel( rand(1) / 104, rand(1) / 164 );
  }
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program assigns different colors to randomly selected pixels.

Summary

#include <graph.h>

void far _setfillmask(*mask*);	
unsigned char far *mask;	Mask array

Description

The **_setfillmask** function sets the current fill mask. The mask is an 8-by-8 array of bits, where each bit represents a pixel. A 1 bit sets the corresponding pixel to the current color, while a 0 bit leaves the pixel unchanged. The mask is repeated over the entire fill area.

If no fill mask is set (*mask* is **NULL**—the default), only the current color is used in fill operations.

Return Value

There is no return value.

See Also

_floodfill, **_getfillmask**

Example

```
#include <stdio.h>
#include <graph.h>

unsigned char *(style[ 6 ]) = { "x00x00x00x00x00x00x00x00",
    "x20x08x20x08x20x08x20x08", "x98xc6x30x30x8cx4cx62x18",
    "xe6x38xb2x9cxe6x38xb2x9c", "xfcxeex7axdexf6xbcxeex7a",
    "xfexfexfexfexfexfexfexfe" };

char *oldstyle = "12345678";   /* place holder for old style */

main()
{
 int loop;
 _setvideomode(_MRES4COLOR);
 _getfillmask( oldstyle );
 _setcolor( 2 ); /* draw an ellipse under the  */
 /* middle few rectangles in a different color */
 _ellipse( _GFILLINTERIOR, 120, 75, 200, 125 );
 _setcolor( 3 );
 for ( loop = 0; loop < 6; loop++ ) {
  /* make 6 rectangles, the first background color */
  _setfillmask( (char far *)(style[ loop ]) );
  _rectangle(_GFILLINTERIOR,loop*40+5,90,(loop+1)*40,110);
 }
 _setfillmask( oldstyle );    /* restore old style */
 while ( !kbhit() );    /* Strike any key to continue */
 _setvideomode (_DEFAULTMODE);
}
```

This program draws an ellipse overlaid with six rectangles, each with a different fill mask, as shown in Figure R.7.

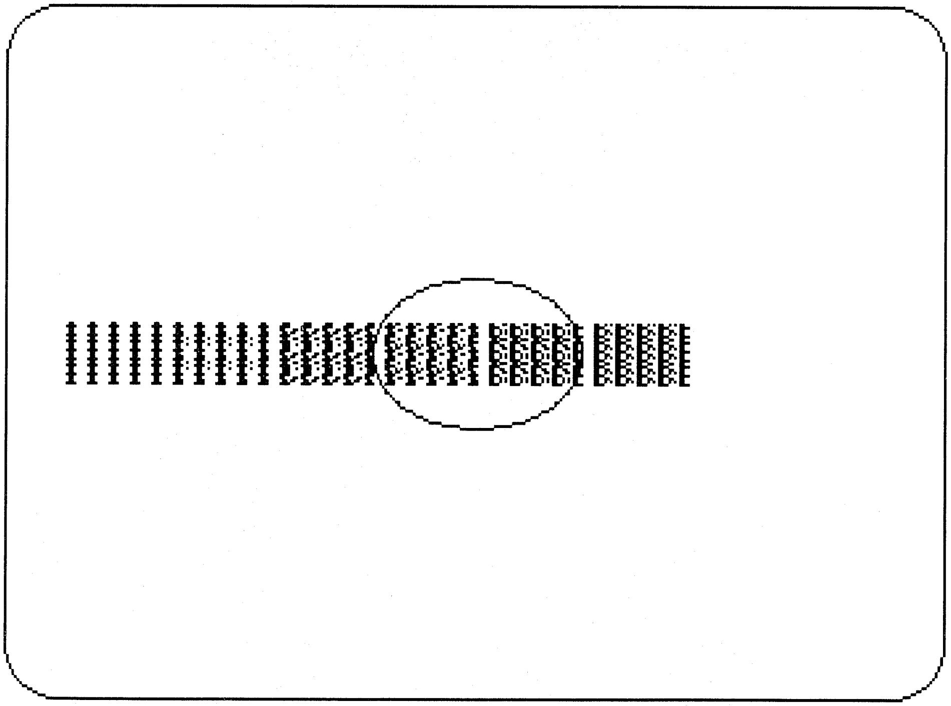


Figure R.7 Output of _setfillmask Program

Summary

#include <setjmp.h>

int setjmp(*env*);	
jmp_buf *env*;	Variable in which environment is stored

Description

The **setjmp** function saves a stack environment that can subsequently be restored using **longjmp**. Used together this way, **setjmp** and **longjmp** provide a way to execute a nonlocal goto. They are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable.

Return Value

The **setjmp** function returns 0 after saving the stack environment. If **setjmp** returns as a result of a **longjmp** call, it returns the value argument of **longjmp**, or, if the *value* argument of **longjmp** is 1, it returns 0. There is no error return.

See Also

longjmp

Warning

The values of register variables in the routine calling **setjmp** may not be restored to the proper values after a **longjmp** call is executed.

Example

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

main( )
        {
        if (setjmp(mark) != 0) {
                printf("longjmp has been called\n");
                recover( );
                exit(1);
                }
        printf("setjmp has been called\n");
        .
        .
        .
        p( );
        .
        .
        .
        }
p( )
        {
        int error = 0;
        .
        .
        .
        if (error != 0)
                longjmp(mark,-1);
        .
        .
        .
        }
recover( )
        {
        /* ensure that data files won't be corrupted by
        ** exiting the program.
        */
        .
        .
        .
        }
```

This program uses **setjmp** to save the stack environment and executes the `p` function to simulate an error. It then uses **longjmp** to restore the stack environment and resume execution immediately after the **setjmp** call. Because **longjmp** and **setjmp** return different values, a conditional expression in the program allows the program to call the `recover` function to use additional error-recovery code.

Summary

#include <graph.h>

void far _setlinestyle(*mask*);
unsigned short *mask*; Desired line-style mask

Description

Some graphics routines (**_lineto** and **_rectangle**) draw straight lines on the screen. The type of line is controlled by the current line-style mask.

The **_setlinestyle** function selects the mask used for line drawing. The mask is a 16-bit array, where each bit represents a pixel in the line being drawn. If a bit is 1, the corresponding pixel is set to the color of the line (the current color). If a bit is 0, the corresponding pixel is left unchanged. The template is repeated for the entire length of the line. The default mask is 0xFFF (a solid line).

See Also

_getlinestyle

Example

```
#include <stdio.h>
#include <graph.h>

short style[16] = {0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff,
    0x1ff, 0x3ff, 0x7ff, 0xfff, 0x1fff, 0x3fff, 0x7fff,
    0xffff};
```

```
main()
{
 int xvar, yvar, loop, oldstyle;
 _setvideomode(_MRES16COLOR);
 oldstyle = _getlinestyle(); /* Save the old style of line */
 for (xvar = 0, loop = 0; xvar < 320; xvar += 3, loop++) {
  _setcolor( loop % 16 );
  yvar = xvar * 5 / 8;
  _setlinestyle( style[ loop % 16 ] );
  _rectangle( _GBORDER, 320 - xvar, 200 - yvar, xvar, yvar );
 }
 _setlinestyle(oldstyle);
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_setlinestyle** to set a new line style for each of a series of rectangles.

Summary

#include <graph.h>

struct xycoord {
  short xcoord; x coordinate
  short ycoord; y coordinate
  } far _setlogorg(*x*, *y*);
short *x*, *y*; New origin point

Description

The **_setlogorg** function moves the logical origin (0, 0) to the physical point (*x*, *y*). All other logical points move the same direction and distance.

Return Value

The function returns the physical coordinates of the previous logical origin in an **xycoord** structure, defined in **graph.h**.

See Also

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
  struct videoconfig config;
 _setvideomode(_MRES16COLOR);
 _getvideoconfig( &config );
 /* Set logical origin to the center of the screen */
 _setlogorg(config.numxpixels/2-1,config.numypixels/2-1);
 _moveto( -80, -50 );
 _lineto( 80, 50 );
 _lineto( 80, -50 );
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_setlogorg** to put the logical origin in the center of the screen.

S

setmode

Summary

#include <fcntl.h>	
#include <io.h>	Required only for function declarations
int setmode(*handle, mode***);**	
int *handle***;**	File handle
int *mode***;**	New translation mode

Description

The **setmode** function sets the translation mode of the file given by *handle* to *mode*. The mode must be one of the following manifest constants:

Constant	Meaning
O_TEXT	Set text (translated) mode. Carriage-return–line-feed combinations (CR-LF) are translated into a single line feed (LF) on input. Line-feed characters are translated into CR-LF combinations on output.
O_BINARY	Set binary (untranslated) mode. The above translations are suppressed.

The **setmode** function is typically used to modify the default translation mode of **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, but can be used on any file.

Return Value

If successful, **setmode** returns the previous translation mode. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EBADF	Invalid file handle
EINVAL	Invalid *mode* argument (neither **O_TEXT** nor **O_BINARY**)

See Also

creat, **fopen**, **open**

Example

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int result;

main()
 {
 /* Set "stdin" to have binary mode    */
 /* (Initially "stdin" is in text mode): */
  result = setmode(fileno(stdin),O_BINARY);
  if ( result == -1 )
    perror("Cannot set mode");
  else
    printf("'stdin' successfully changed to binary mode");
 }
```

This program uses **setmode** to change **stdin** from text mode to binary mode.

_setpixel

Summary

#include <graph.h>

short far _setpixel(*x*, *y*);
short *x*, *y*; Target pixel

Description

The **_setpixel** function sets the pixel at the logical point (*x*, *y*) to the current color.

Return Value

The function returns the previous value of the target pixel. If the function fails (for example, the point lies outside of the clipping region), it will return –1.

See Also

_getpixel

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop;
 int xvar, yvar;
 _setvideomode(_MRES16COLOR);
 _rectangle(_GFILLINTERIOR, 80, 50, 240, 150 );
 for (loop = 0; loop < 8000L; loop++) {
  /* Fill pixels at random, but only if they are already on */
  if (_getpixel(xvar=rand(1) / 104, yvar=rand(1) / 164)) {
   _setcolor(rand(1) % 16);
   _setpixel(xvar, yvar);
  }
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program draws a rectangle and assigns different colors to randomly selected pixels.

Summary

#include <graph.h>

short far _settextcolor(*pixel*);
short *pixel*; Desired pixel value

Description

The **_settextcolor** function sets the current text color parameter to the pixel value specified by *pixel*. The default text color is the same as the maximum pixel value.

In text color mode, you can specify a pixel value in the range 0–31. The colors in the range 0–15 are interpreted as normal; colors in the range 16–31 are the same as those in the range 0–15, but also have blinking text. The normal color range is defined below:

No.	Color	No.	Color
0	Black	8	Dark gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	Bright white

Note

Text colors are not restricted to the current palette, as are the current color and current background color.

Return Value

The function returns the pixel value of the previous text color. There is no error return.

See Also

_gettextcolor

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];

main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);      /* Turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on last line; is out of the window");
 while (!kbhit());    /* wait */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_settextcolor** to change from the default text color prior to outputting text, and again at the end of the text output to restore the previous color.

Summary

#include <graph.h>

struct rccoord {	
short row;	Row coordinate
short col;	Column coordinate
} far _settextposition(*row, column*);	
short *row, column*;	New output start position

Description

This function relocates the current text position to the display point (*row, column*). Subsequent text output produced with the **_outtext** function (as well as standard console I/O routines, such as **printf**) proceeds from that point.

Return Value

The function returns the previous text position in an **rccoord** structure, defined in **graph.h**.

See Also

_gettextposition, **_outtext**

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];
```

```
main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);     /* Turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on last line; is out of the window");
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_settextposition** several times to start text output in different places on the screen.

Summary

#include <graph.h>

void far _settextwindow(*r1*, *c1*, *r2*, *c2***);**

short *r1*, *c1*;	Upper-left corner of window
short *r2*, *c2*;	Lower-right corner of window

Description

The **_settextwindow** function specifies a window in row and column coordinates where all the text output to the screen is displayed. The arguments (*r1*, *c1*) specify the upper-left corner of the window, and the arguments (*r2*, *c2*) specify the lower-right corner of the window.

Text is output from the top of the window down. When the text window is full, the uppermost line scrolls up out of it.

Return Value

There is no return value.

See Also

_outtext

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];
```

```
main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);      /* Turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on last line; is out of the window");
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program creates a text window in the upper half of the screen, then writes text to it.

Summary

#include <stdio.h>

int setvbuf(*stream, buffer, type, size*);	
FILE **stream*;	Pointer to **FILE** structure
char **buffer*;	User-allocated buffer
int *type*;	Type of buffer: **_IOFBF** (full buffering) **_IOLBF** (line buffering) **_IONBF** (no buffer)
size_t *size*;	Size of buffer

Description

The **setvbuf** function allows the user to control both buffering and buffer size for *stream*. The *stream* must refer to an open file which has been read or written to since being opened. The array that *buf* points to is used as the buffer, unless it is **NULL**, in which case an automatically allocated buffer *size*-bytes long is used. The type must be **_IOLBF**, **_IOFBF**, or **_IONBF**. If *type* is **_IOFBF** or **_IOLBF**, then *size* is used as the size of the buffer. If *type* is **_IONBF**, then the stream is unbuffered, and *size* and *buf* are ignored.

Type	Meaning
_IOFBF	Full buffering; that is, *buffer* is used as the buffer and *size* is used as the size of the buffer. If *buffer* is **NULL**, an automatically allocated buffer *size* bytes long is used.
_IOLBF	Same as **_IOFBF**.
_IONBF	No buffer is used, regardless of *buffer* or *size*.

C 4.0 Difference

Under Version 4.0 of Microsoft C, no buffering occurs if *buffer* is **NULL**.

The legal values for *size* are greater than 0 and less than the maximum integer size.

Return Value

The return value for **setvbuf** is 0 if successful, and nonzero if an illegal type or buffer size is specified.

See Also

setbuf, **fclose**, **fflush**, **fopen**

Example

```
#include <stdio.h>
char buf[1024];
FILE *stream1, *stream2;
int result;

main()
 {
  stream1 = fopen("data1","r");
  stream2 = fopen("data2","w");
  if (result = setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
    printf("Incorrect type or size of buffer1\n");
  else
    printf("'stream1' now has a buffer of 1024 bytes\n");
  if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
    printf("Incorrect type or size of buffer1\n");
  else
    printf("'stream2' now has no buffer\n");
 }
```

This program opens two streams named `stream1` and `stream2`. It then uses **setvbuf** to give `stream1` a user-defined buffer of 1024 bytes and `stream2` no buffer at all.

Summary

#include <graph.h>

short far _setvideomode(*mode*);
short *mode*; Desired mode

Description

The **_setvideomode** function selects a screen mode appropriate for a particular hardware/display configuration. The *mode* argument can be one of the manifest constants shown in Table R.8 (defined in **graph.h**).

Table R.8
Manifest Constants for Screen Mode

Mode	Type[1]	Size[2]	Colors[3]	Adapter[4]
_DEFAULTMODE		Hardware default mode		
_TEXTBW40	M/T	40x25	16	CGA
_TEXTC40	C/T	40x25	16	CGA
_TEXTBW80	M/T	80x25	16	CGA
_TEXTC80	C/T	80x25	16	CGA
_MRES4COLOR	C/G	320x200	4	CGA
_MRESNOCOLOR	M/G	320x200	4	CGA
_HRESBW	M/G	640x200	2	CGA
_TEXTMONO	M/T	80x25	1	MA
_MRES16COLOR	C/G	320x200	16	EGA
_HRES16COLOR	C/G	640x200	16	EGA
_ERESNOCOLOR	M/T	640x350	1	EGA
_ERESCOLOR	C/G	640x350	64	EGA
_VRES2COLOR	C/G	640x480	2	VGA
_VRES16COLOR	C/G	640x480	16	VGA
_MRES256COLOR	C/G	320x200	256	VGA

[1] M indicates monochrome, C indicates color output, T indicates text, and G indicates graphics generation.

[2] For text modes, size is given in characters (columns x rows). For graphics modes, size is given in pixels (horizontal x vertical).

[3] For monochrome displays, the number of colors is the number of gray shades.

[4] Adapters are the IBM (and compatible) Monochrome Adapter (MA), Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), and Video Graphics Array (VGA).

Note

Only IBM hardware is described here, but display hardware that is strictly compatible with IBM hardware should also work as described.

■ Return Value

The function returns a nonzero value if the function is successful. If an error is encountered (that is, the mode selected is not supported by the current hardware configuration), the function returns 0.

■ See Also

_getvideoconfig

■ Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 struct videoconfig config;
 _setvideomode(_MRES16COLOR);
 _getvideoconfig( &config );
 /* set logical origin to the center of the screen */
 _setlogorg(config.numxpixels/2-1,config.numypixels/2-1);
 _moveto( -80, -50 );
 _lineto( 80, 50 );
 _lineto( 80, -50 );
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_setvideomode** to set the video mode **_MRES16COLOR** (320 x 200 color graphics).

Summary

#include <graph.h>

void far _setviewport(*x1, y1, x2, y2*);
short *x1, y1*; Upper-left corner of window
short *x2, y2*; Lower-right corner of window

Description

The **_setviewport** function defines a clipping region in exactly the same manner as **_setcliprgn**, and then sets the logical origin to the upper-left corner of the region. The physical points (*x1, y1*) and (*x2, y2*) are the diagonally opposed corners of the rectangular clipping region.

Return Value

There is no return value.

See Also

_setlogorg, **_setcliprgn**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
    _setvideomode(_MRES16COLOR);
    _setviewport(0, 0, 200, 125 );
    _ellipse(_GFILLINTERIOR, 80, 50, 240, 150 );
    /* wait for key before resetting screen */
    while (!kbhit());
    _setvideomode (_DEFAULTMODE);
}
```

This program sets the viewport in the upper-left quadrant of the screen, then draws an ellipse lying partly outside the viewport.

_setvisualpage

Summary

#include <graph.h>

short far _setvisualpage(*page*);
short *page*; Visual page number

Description

For hardware configurations that include an EGA and enough memory to support multiple-screen pages, the **_setvisualpage** function selects the current visual page. The *page* argument specifies the current visual page. The default page number is 0.

Return Value

The function returns the page number of the previous visual page. If the function fails, it returns a negative value.

See Also

_setactivepage, **_setvideomode**

Example

```
#include <stdio.h>
#include <graph.h>

main()
{
 int loop = 0;
 _setvideomode( _MRES16COLOR );
 while ( !kbhit() ) { /* Repeat until a character is typed */
  /* alternate between page 0 and page 1 */
  _setactivepage( loop & 1 );
  _setcolor( loop % 16 );
  _rectangle( _GFILLINTERIOR, 80, 50, 240, 150 );
  _setvisualpage( loop++ & 1 );
 }
 _setvideomode (_DEFAULTMODE);
}
```

This program sets a visual page and an active page and displays them alternately.

Summary

#include <signal.h>

void (*signal(*sig*, *func*(*sig*⟦, *subcode*⟧)))(*sig*);

int *sig*;	Signal to be mapped
void **func*();	Function to be executed on *sig*
int *subcode*;	Optional error subcode

Description

The **signal** function allows a process to define the interrupt handler signals from the operating system.

The *sig* argument must be one of the manifest constants listed below (defined in **signal.h**):

Signal	Meaning
SIGABRT	Abnormal termination. The default action terminates the calling program with exit code 3.
SIGFPE	Floating-point error, such as overflow, division by zero, or invalid operation. The default action terminates the calling program.
SIGILL	Illegal instruction. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action terminates the calling program.
SIGINT	CTRL+C interrupt. The default action issues INT 0x23.
SIGSEGV	Illegal storage access. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action terminates the calling program.
SIGTERM	Termination request sent to the program. This signal is not generated by MS-DOS, but is supported for ANSI compatibility. The default action terminates the calling program.

The *func* argument must be one of the manifest constants **SIG_DFL** or **SIG_IGN** (also defined in **signal.h**), or a symbolic or assembly-language function address. The action taken in response to the interrupt signal depends on the value of *func*, as follows:

Table R.9

Function Arguments

Value	Meaning
SIG_DFL	The default action is taken. If the calling process is terminated, control returns to the MS-DOS command level; all files opened by the process are closed, but buffers are not flushed.
SIG_IGN	The interrupt signal is ignored. This value should never be given for **SIGFPE**, since the floating-point state of the process is left undefined.
Function address	For **SIGINT** signals, the function pointed to by *func* is passed the single argument **SIGINT** and executed. If the function returns, the calling process resumes execution immediately following the point where it received the interrupt signal. Before the specified function is executed, the value of *func* is set to **SIG_DFL**; the next interrupt signal is treated as described above for **SIG_DFL**, unless an intervening call to **signal** specifies otherwise. This allows the user to reset signals in the called function if desired. For **SIGFPE**, the function pointed to by *func* is passed two arguments (**SIGFPE** and an integer error subcode, **FPE_***xxx*), then executed. (See the include file **float.h** for definitions of the **FPE_***xxx* subcodes.) The second value is not part of the ANSI standard; it is a Microsoft extension. The value of *func* is not reset upon receiving the signal; to recover from floating-point exceptions, use **setjmp** in conjunction with **longjmp**. (See the example under **_fpreset** in this Reference.) If the function returns, the calling process resumes execution with the floating-point state of the process left in an undefined state.

Return Value

The **signal** function returns the previous value of *func*. A return value of **SIG_ERR** indicates an error, and **errno** is set to **EINVAL**, indicating an invalid *sig* value.

See Also

abort, **exec** functions, **exit**, **_exit**, **_fpreset**, **spawn** functions

---

Note

> Signal settings are not preserved in child processes created by calls to **exec** or **spawn** routines. The signal settings are reset to the default in the child process.

Example

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

int handler();

main()
 {
 /* Set so that interrupt calls "handler": */
  if (signal(SIGINT,handler) == (int(*)())-1) {
    fprintf(stderr,"Couldn't set SIGINT");
    abort();
   }
  for(;;)printf("Hit control C:\n");
 }

int handler() /* Function called at OS interrupt */
 {
  char ch;

  /* Disallow ctrl-c during handler: */
  signal(SIGINT, SIG_IGN);
  printf("Terminate processing? ");
  ch = getch();
  if (( ch == 'y' ) ¦¦ ( ch == 'Y')) exit(0);
  /* "signal" called here so next         */
  /* interrupt signal sends control       */
  /* to handler(), not to operating sys.: */
  signal(SIGINT,handler);
 }
```

This program uses **signal** to set up the `handler` function as the routine that is called to execute an operating-system interrupt. When the user presses CTRL+C, `handler` is called to handle the interrupt.

Summary

#include <math.h>

double sin(*x*);	Calculates sine of *x*
double sinh(*x*);	Calculates hyperbolic sine of *x*
double *x*;	Angle in radians

Description

The **sin** and **sinh** calls find the sine and hyperbolic sine of *x*, respectively.

Return Value

The **sin** function returns the sine of *x*. If *x* is large, a partial loss of significance in the result may occur, and **sin** generates a **PLOSS** error. If *x* is so large that significance is completely lost, **sin** prints a **TLOSS** message to **stderr** and returns 0. In both cases, **errno** is set to **ERANGE**.

The **sinh** function returns the hyperbolic sine of *x*. If the result is too large, **sinh** sets **errno** to **ERANGE** and returns ±**HUGE_VAL**.

See Also

acos, **asin**, **atan**, **atan2**, **cos**, **cosh**, **tan**, **tanh**

Example

```
#include <math.h>
#include <stdio.h>

main()
{
 double pi = 3.1415926535;
 double x = pi/2;
 double y = sin(x);                /* y is 1.0 */
 printf("The sin(%f) = %f\n",x,y);
 y = sinh(x);                 /* y is 2.301299 */
 printf("The sinh(%f) = %f\n",x,y);
}
```

This program displays the sine and hyperbolic sine of $\pi/2$.

sopen

Summary

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>                    Required only for function declarations

int sopen(path, oflag, shflag⟦, pmode⟧);
char *path;                        File path name
int oflag;                         Type of operations allowed
int shflag;                        Type of sharing allowed
int pmode;                         Permission setting
```

Description

The **sopen** function opens the file specified by *path* and prepares the file for subsequent shared reading or writing, as defined by *oflag* and *shflag*. The integer expression *oflag* is formed by combining one or more of the following manifest constants, defined in **fcntl.h**. When more than one manifest constant is given, the constants are joined with the OR operator (¦).

Constant	Meaning
O_APPEND	Repositions the file pointer to the end of the file before every write operation.
O_BINARY	Opens file in binary (untranslated) mode. (See **fopen** for a description of binary mode.)
O_CREAT	Creates and opens a new file. This has no effect if the file specified by *path* exists.
O_EXCL	Returns an error value if the file specified by *path* exists. This applies only when used with **O_CREAT**.
O_RDONLY	Opens file for reading only. If this flag is given, neither the **O_RDWR** flag nor the **O_WRONLY** flag can be given.
O_RDWR	Opens file for both reading and writing. If this flag is given, neither **O_RDONLY** nor **O_WRONLY** can be given.
O_TEXT	Opens file in text (translated) mode. (See **fopen** for a description of text mode.)

O_TRUNC	Opens and truncates an existing file to 0 bytes. The file must have write permission; the contents of the file are destroyed.
O_WRONLY	Opens file for writing only. If this flag is given, neither **O_RDONLY** nor **O_RDWR** can be given.

Note

O_TRUNC destroys the entire contents of an existing file. Use it with care.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in **share.h**. If **SHARE.COM** (or **SHARE.EXE** for some versions of MS-DOS) is not installed, MS-DOS ignores the sharing mode. (See your system documentation for detailed information about sharing modes.)

Constant	Meaning
SH_COMPAT	Sets compatibility mode
SH_DENYRW	Denies read and write access to file
SH_DENYWR	Denies write access to file
SH_DENYRD	Denies read access to file
SH_DENYNO	Permits read and write access

The *pmode* argument is required only when **O_CREAT** is specified. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. Otherwise, the *pmode* argument is ignored. The *pmode* argument is an integer expression that contains one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in **sys\stat.h**. When both constants are given, they are joined with the OR operator (¦). The meaning of the *pmode* argument is as follows:

Value	Meaning
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD ¦ S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read only. Under MS-DOS, all files are readable; it is not possible to give write-only permission. Thus the modes **S_IWRITE** and **S_IREAD ¦ S_IWRITE** are equivalent.

Important

Under MS-DOS Versions 3.0, 3.1, and 3.2 with **SHARE** installed, a problem occurs when opening a new file with **sopen** under the following sets of conditions:

- With *oflag* set to **O_CREAT ¦ O_RDONLY** or **O_CREAT ¦ WRONLY**, *pmode* set to **S_IREAD**, and *shflag* set to **SH_COMPAT**
- With *oflag* set to any combination that includes **O_FLAG**, *pmode* set to **S_IREAD**, and *shflag* set to anything other than **SH_COMPAT**

In either case, the operating system will prematurely close the file during system calls made within **sopen**, or the system will generate a sharing violation (INT 24H).

To avoid the problem, open the file with *pmode* set to **S_IWRITE**. After closing the file, call **chmod** and change the mode back to **S_IREAD**. Another way around the problem is to open the file with *pmode* set to **S_IREAD**, *oflag* set to **O_CREAT ¦ O_RDWR**, and *shflag* set to **SH_COMPAT**.

The **sopen** function applies the current file-permission mask to *pmode* before setting the permissions (see **umask**).

Return Value

The **sopen** function returns a file handle for the opened file. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Given path name is a directory; or the file is read only but an open for writing was attempted; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; MS-DOS Versions 3.0 and later only).
EEXIST	The **O_CREAT** and **O_EXCL** flags are specified, but the named file already exists.
EMFILE	No more file handles available (too many open files).
ENOENT	File or path name not found.

See Also

close, **creat**, **fopen**, **open**, **umask**

Note

The **sopen** function should be used only under MS-DOS Versions 3.0 and later. Under earlier versions of MS-DOS, the *shflag* argument is ignored.

File-sharing modes will not work correctly for buffered files, so do not use **fdopen** to associate a file opened for sharing (or locking) with a stream.

Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>

extern unsigned char _osmajor;
int fh;

main()
 {

   /* Open for file sharing: */
   if (_osmajor >= 3)
     fh = sopen("sopen.c", O_RDWR | O_BINARY, SH_DENYRW);

   /* Just a regular open */
   else
     fh =  open("sopen.c", O_RDWR | O_BINARY );

   if (fh == -1)
     perror("Failure on an attempt to open the file\n");
   else
     printf("File opened successfully");

   if (_osmajor >= 3)
     printf("At least version 3.0, sopen used.\n");
   else
     printf("Pre version 3.0, open used.\n");
 }
```

This program first checks the version of MS-DOS. If the version is 3.0 or later, it uses **sopen** to open a file named `sopen.c` for sharing.

Summary

#include <stdio.h>
#include <process.h>

int spawnl(*modeflag, path, arg0, arg1..., argn,*NULL);

int spawnle(*modeflag, path, arg0, arg1..., argn,*NULL, *envp*);

int spawnlp(*modeflag, path, arg0, arg1..., argn,*NULL);

int spawnlpe(*modeflag, path, arg0, arg1..., argn,*NULL, *envp*);

int spawnv(*modeflag, path, argv*);

int spawnve(*modeflag, path, argv, envp*);

int spawnvp(*modeflag, path, argv*);

int spawnvpe(*modeflag, path, argv, envp*);

int *modeflag*;	Execution mode for parent process
char **path*;	Path name of file to be executed
char **arg0*,**arg1*,...,**argn* ;	List of pointers to arguments
char **argv*[ ];	Array of pointers to arguments
char **envp*[ ];	Array of pointers to environment settings

Description

The **spawn** functions create and execute a new child process. Enough memory must be available for loading and executing the child process. The *modeflag* argument determines the action taken by the parent process before and during the **spawn**.

All of the functions in this family use the same basic **spawn** function; the letter(s) at the end of the function name specifies the particular variation:

Letter	Variation
p	Uses the **PATH** environment variable to find the file to be executed
l	Lists command-line arguments separately

v	Passes the child process an array of pointers to command-line arguments
e	Passes the child process an array of pointers to environment arguments

The following values for *modeflag* are defined in **process.h**:

Value	Meaning
P_WAIT	Suspends parent process until execution of child process is complete (synchronous **spawn**)
P_NOWAIT	Continues to execute parent process concurrently with child process
P_OVERLAY	Overlays parent process with child, destroying the parent (same effect as **exec** calls)

Only **P_WAIT** and **P_OVERLAY** may be used for *modeflags*. The **P_NOWAIT** value is reserved for future implementation. An error value is returned if **P_NOWAIT** is used.

The *path* argument specifies the file to be executed as the child process. The path can be a full path (from the root), a partial path (from the current working directory), or just a file name. If *path* does not have a filename extension or end with a period (.), **spawn** will search for the file; if unsuccessful, the extension **.EXE** is attempted. If *path* has an extension, only that extension is used. If *path* ends with a period, the **spawn** calls search for *path* with no extension. The **spawnlp**, **spawnlpe**, **spawnvp**, and **spawnvpe** routines search for *path* (using the same procedures) in the directories specified by the **PATH** environment variable.

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the **spawn** call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes. The terminating null character (**'\0'**) for each string is not included in the count, but space characters (inserted automatically to separate the arguments) are included.

The argument pointers can be passed as separate arguments (**spawnl**, **spawnle**, **spawnlp**, and **spawnlpe**) or as an array of pointers (**spawnv**, **spawnve**, **spawnvp**, and **spawnvpe**). At least one argument, *arg0*, must be passed to the child process (which sees it as *argv*[**0**]). Usually, this argument is a copy of the *path* argument. (A different value will not produce an error.) Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* is not available for use in the child process. However, under MS-DOS versions 3.0 and later, the path is available as *arg0*.

The **spawnl**, **spawnle**, **spawnlp**, and **spawnlpe** calls are typically used in cases where the number of arguments is known in advance. The *arg0* argument is usually a pointer to *path*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a null argument list.

The **spawnv**, **spawnve**, **spawnvp**, and **spawnvpe** calls are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. The argument *argv*[**0**] is usually a pointer to *path*, and *argv*[**1**] through *argv*[*n*] are pointers to the character strings forming the new argument list. The argument *argv*[*n*+**1**] must be a null pointer to mark the end of the argument list.

Files that are open when a **spawn** call is made remain open in the child process. In the **spawnl**, **spawnlp**, **spawnv**, and **spawnvp** calls, the child process inherits the environment of the parent. The **spawnle**, **spawnlpe**, **spawnve**, and **spawnvpe** calls allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. The argument *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

name=value

where *name* is an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotes.) The final element of the *envp* array should be **NULL**. When *envp* itself is **NULL**, the child process inherits the environment settings of the parent process.

The **spawn** functions pass the child process all information about open files, including the translation mode, through the **;C_FILE_INFO** entry in the environment that is passed. The C start-up code normally processes this entry and then deletes it from the environment. However, if a **spawn** function spawns a non-C process (such as **COMMAND.COM**), this entry will remain in the environment. In this case, since the environment information is passed in binary form, printing the environment will show graphics characters in the definition string for this entry. It has no other effect on normal operations.

Return Value

The return value from a synchronous **spawn** (**P_WAIT** specified for *modeflag*) is the exit status of the child process.

The return value from an asynchronous **spawn** (**P_NOWAIT** specified for *modeflag*) is the process ID. To obtain the exit code for the spawned process, you must call the **wait** or **cwait** function and specify the process ID.

The exit status is 0 if the process terminated normally. The exit status can be set to a nonzero value if the child process specifically calls the **exit** routine with a nonzero argument. If the child process does not set a positive exit status, the positive exit status indicates an abnormal exit with an **abort** call or an interrupt.

A return value of –1 indicates an error (the child process is not started). In this case, **errno** is set to one of the following values:

Value	Meaning
E2BIG	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EINVAL	The *modeflag* argument is invalid.
ENOENT	The file or path name was not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the child process.

Note

Signal settings are not preserved in child processes created by calls to **spawn** routines. The signal settings are reset to the default in the child process.

■ See Also

abort, **atexit**, **exec** functions, **exit**, **_exit**, **onexit**, **system**

Example

```
#include <stdio.h>
#include <process.h>


char *my_env[] = {
                  "THIS=environment will be",
                  "PASSED=to child.exe by the",
                  "SPAWNLE=and",
                  "SPAWNLPE=and",
                  "SPAWNVE=and",
                  "SPAWNVPE=functions",
                  NULL
                  };

main(argc, argv)
int argc;
char *argv[];
 {
  char *args[4];
  int result;

  args[0] = "child";    /* Set up parameters to send */
  args[1] = "spawn??";
  args[2] = "two";
  args[3] = NULL;

  switch (argv[1][0])   /* Based on first letter of argument */
   {
    case '1':
      spawnl  (P_WAIT, "child.exe","child ","spawnl",
                                        "two",NULL);
    break;
    case '2':
      spawnle (P_WAIT, "child.exe","child","spawnle",
                                "two",NULL,my_env);
    break;
    case '3':
      spawnlp (P_WAIT, "child.exe","child","spawnlp",
                                        "two",NULL);
    break;
    case '4':
      spawnlpe(P_WAIT, "child.exe","child","spawnlpe",
                                "two",NULL,my_env);
    break;
    case '5':
      spawnv  (P_OVERLAY, "child.exe",args);
    break;
    case '6':
      spawnve (P_OVERLAY, "child.exe",args,my_env);
    break;
    case '7':
      spawnvp (P_OVERLAY, "child.exe",args);
    break;
    case '8':
      spawnvpe(P_OVERLAY, "child.exe",args,my_env);
    break;
    default:
      printf("Enter a number from 1 to 8 as a command
      line parameter.");
      exit(0);
   }

 printf("\n\nReturned from SPAWN!\n");
 }
```

This program accepts a number in the range 1 – 8 from the command line. Based on the number it receives, it executes one of the eight different procedures that spawn the process named `child`. For some of these procedures, the `child.exe` file must be in the same directory; for others, it must only be in the same path.

_splitpath

Summary

#include <stdlib.h>

void _splitpath(*path*, *drive*, *dir*, *fname*, *ext*);

char ******path*;	Full path-name buffer
char ******drive*;	Drive letter
char ******dir*;	Directory path
char ******fname*;	File name
char ******ext*;	File extension

Description

The **_splitpath** routine decomposes an existing path name into the four components. The *path* argument should point to a buffer containing the complete path name. The maximum size necessary for each buffer is specified by the **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_NAME**, and **_MAX_EXT** manifest constants defined in **stdlib.h**. The other arguments point to the following buffers used to store the path-name elements:

Buffer	**Description**
drive	Contains the drive letter followed by a colon (:) if a drive is specified in *path*.
dir	Contains the path of subdirectories, if any, including the trailing slash. Forward slashes (/), backslashes (\\), or both may be present in *path*.
fname	Contains the base file name without any extensions.
ext	Contains the file-name extension, if any, including the leading period (.).

The return parameters will contain empty strings for any path-name components not found in *path*.

Example

```
#include <dos.h>

main()
 {
  char path_buffer [40];
  char * drive [3];
  char * dir [30];
  char * fname [9];
  char * ext [4];

  _makepath (path_buffer, "c", "qc\\clibref\\", "makepath", "c");
  printf ("path created with _makepath: %s\n\n", path_buffer);

  _splitpath (path_buffer, drive, dir, fname, ext);
  printf ("path extracted with _splitpath\n");
  printf ("drive: %s\n", drive);
  printf ("dir: %s\n", dir);
  printf ("fname: %s\n", fname);
  printf ("ext: %s\n", ext);
 }
```

This program builds a file-name path from the specified components, then extracts the individual components.

Summary

#include <stdio.h>

int sprintf(*buffer*, *format*⟦, *argument*⟧...);

char **buffer*;	Storage location for output
const char **format*;	Format-control string

Description

The **sprintf** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for the **printf** function; see the **printf** reference page for a description of the format and arguments. A **NULL** is appended to the end of the characters written but is not counted in the return value.

Return Value

The **sprintf** function returns the number of characters stored in *buffer*, not counting the terminating **NULL**.

See Also

fprintf, **printf**, **sscanf**

Example

```
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "computer";
char c;
```

```
main()
 {
  c = 'l';
  i = 35;
  fp =1.7320508;

  /* Format and print various data: */
  j = sprintf(buffer, "%s\n", s);
  j += sprintf(buffer+j, "%c\n", c);
  j += sprintf(buffer+j, "%d\n", i);
  j += sprintf(buffer+j, "%f\n", fp);

  printf("string:\n%s\ncharacter count = %d\n", buffer, j );
 }
```

This program uses **sprintf** to format various data and place them in the string named `buffer`.

Summary

#include <math.h>

double sqrt(*x*);
double *x*; Non-negative floating-point value

Description

The **sqrt** function calculates the square root of *x*.

Return Value

The **sqrt** function returns the square-root result. If *x* is negative, the function prints a **DOMAIN** error message to **stderr**, sets **errno** to EDOM, and returns 0.

Error handling can be modified by using the **matherr** routine.

See Also

exp, **log**, **matherr**, **pow**

Example

```
#include <math.h>
#include <stdio.h>

main()
 {
  double x, y, z;

  x = 1.0;
  y = 3.0;

  if ((z = sqrt(x+y)) == 0.0)  /* Return of 0 means arg<0 */
    {                          /* z is 2                 */
     if ((x+y) < 0.0)
       perror("sqrt of a negative number");
    }
  else
    printf("The square root of %f = %f\n", x+y, z);
 }
```

This program uses **sqrt** to display the square root of 4.

srand

Summary

#include <stdlib.h>	Required only for function declarations
void srand(*seed*);	
unsigned *seed*;	Seed for random-number generation

Description

The **srand** function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the *seed* argument. Any other value for *seed* sets the generator to a random starting point.

The **rand** function is used to retrieve the pseudorandom numbers generated. Calling **rand** before any call to **srand** will generate the same sequence as calling **srand** with *seed* passed as 1.

Return Value

There is no return value.

See Also

rand

Example

```
#include <stdlib.h>
#include <stdio.h>

main()
 {
  int x, ranvals[20];

  srand(17);
  /* Initialize array and output values: */
  for (x = 0; x < 20; ranvals[x++] = rand())
    printf("Iteration %d, ranvals[%d] =%d\n",x+1,x,
            ranvals[x]);
 }
```

First, this program calls **srand** with a value other than 1 to randomize a random-value sequence. Then it initializes an array named `ranvals` with 20 random values.

sscanf

Summary

#include <stdio.h>

int sscanf(*buffer, format*[[**,** *argument*]]...**);**	
const char **buffer***;**	Stored data
const char **format***;**	Format-control string

Description

The **sscanf** function reads data from *buffer* into the locations given by each argument. Every argument must be a pointer to a variable with a type that corresponds to a type specifier in *format.* The format controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function; see the **scanf** reference page for a complete description of *format.*

Return Value

The **sscanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-string. A return value of 0 means that no fields were assigned.

See Also

fscanf, scanf, sprintf

Example

```
#include <stdio.h>

char *tokenstring = "15 12 14...";
int i;
float fp;
char s[81];
char c;

main()
 {
  /* Input various data from tokenstring: */
  sscanf(tokenstring, "%s", s);
  sscanf(tokenstring, "%c", &c);
  sscanf(tokenstring, "%d", &i);
  sscanf(tokenstring, "%f", &fp);
                                    /* Output the data read */
  printf( "string =%s\n", s );     /* s is 15 */
  printf( "character =%c\n", c );  /* c is 1  */
  printf( "integer =%d\n", i );    /* i is 15 */

  /* fp is 15.000000 */
  printf( "floating point number =%f\n", fp );

}
```

This program uses **sscanf** to read data items from a string named `tokenstring`, then displays them.

stackavail

Summary

#include <malloc.h> Required only for function declarations

size_t stackavail(void);

Description

The **stackavail** function returns the approximate size in bytes of the stack space available for dynamic memory allocation with **alloca**.

Return Value

The **stackavail** function returns the size in bytes as an unsigned integer value.

See Also

alloca, **freect**, **memavl**

Example

```
#include <malloc.h>

main()

{
  char *ptr;

  printf("Stack memory available before alloca = %u\n",
          stackavail( ));
  ptr = alloca(1000*sizeof(char));
  printf("Stack memory available after alloca = %u\n",
          stackavail( ));
}
```

Sample output:

```
Stack memory available before alloca = 1682
Stack memory available after alloca = 678
```

This program uses **stackavail** to determine the amount of free space available on the stack. It then allocates memory from the stack and calls **stackavail** again to display the new amount of available free space.

Summary

```
#include <sys\types.h>
#include <sys\stat.h>

int stat(path, buffer);
char *path;            Path name of existing file
struct stat {          Structure to store results:
        dev_t st_dev;
        ino_t st_ino;
        unsigned short st_mode;
        short st_nlink;
        short st_uid;
        short st_gid;
        dev_t st_rdev;
        off_t st_size;
        time_t st_atime;
        time_t st_mtime;
        time_t st_ctime;
        } *buffer;
```

Description

The **stat** function obtains information about the file or directory specified by *path* and stores it in the structure pointed to by *buffer*. The **stat** structure, defined in **sys\stat.h**, contains the following fields:

Field	Value
st_mode	Bit mask for file-mode information. **S_IFDIR** bit set if *path* specifies a directory; **S_IFREG** bit set if *path* specifies an ordinary file. User read/write bits set according to the file's permission mode; user execute bits set according to the file-name extension.
st_dev	Drive number of the disk containing the file (same as **st_dev**).
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.

st_size	Size of the file in bytes.
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).

There are three additional fields in the **stat** structure type that do not contain meaningful values under MS-DOS.

Return Value

The **stat** function returns the value 0 if the file-status information is obtained. A return value of –1 indicates an error, and **errno** is set to **ENOENT**, indicating that the file name or path name can not be found.

See Also

access, **fstat**

Note

If *path* refers to a device, the size and time fields in the **stat** structure are not meaningful.

Example

```
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";
```

```
main()
 {
  /* Get data associated with "data": */
  result = stat("data",&buf);

  /* Check if statistics are valid: */
  if (result != 0)
    perror("Problem getting information ");
  /* Output some of the statistics: */
  else
   {
    printf("File size     : %ld\n",buf.st_size);
    printf("Drive number  : %d\n",buf.st_dev);
    printf("Time modified : %s",ctime(&buf.st_atime));
   }
 }
```

This program uses **stat** to report the size, drive number, and last modification time for the file named `data`.

_status87

Summary

#include <float.h>

unsigned int _status87( );	Gets floating-point status word

Description

The **_status87** function gets the floating-point status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

Return Value

The bits in the value returned indicate the floating-point status. See the **float.h** include file for a complete definition of the bits returned by **_status87**.

Note

> Many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from **_clear87** and **_status87** become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

See Also

_clear87, **_control87**

■ Example

```
#include <stdio.h>
#include <float.h>

double a = 1e-40, b;
float x, y;

main()
 {
  printf("Status = %.4x - clear\n",_status87());

  /* Store into y is inexact & underflows: */
  y = a;
  printf("Status = %.4x - inexact, underflow\n",_status87());

  /* y is denormal: */
  b = y;
  printf("Status = %.4x - inexact underflow, denormal\n",
          _status87());

  /* Clear user 8087: */
  _clear87();
 }
```

This program creates various floating-point errors and then uses **_status87** to display messages indicating these problems.

Summary

#include <string.h>	Required only for function declarations
char *strcat(string1, string2);	Appends *string2* to *string1*
char *string1;	Destination string
const char *string2;	Source string
char *strchr(string, c);	Searches for first occurrence of *c* in *string*
const char *string;	Source string
int c;	Character to be located
int strcmp(string1, string2);	Compares strings
const char *string1;	
const char *string2;	
int strcmpi(string1, string2);	Compares strings without regard to case
const char *string1;	
const char *string2;	
char *strcpy(string1, string2);	Copies *string2* to *string1*
char *string1;	Destination string
const char *string2;	Source string
size_t strcspn(string1, string2);	Finds first substring in *string1* of characters not in *string2*
const char *string1;	Source string
const char *string2;	Character set
char *strdup(string);	Duplicates *string*
const char *string;	Source string
int stricmp(string1, string2);	Compares strings without regard to case
const char *string1;	
const char *string2;	

Description

The **strcat**, **strchr**, **strcmp**, **strcmpi**, **strcpy**, **strcspn**, **strdup**, and **stricmp** functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character (**'\0'**) marking the end of the string. No overflow checking is performed when strings are copied or appended.

The **strcat** function appends *string2* to *string1*, terminates the resulting string with a null character, and returns a pointer to the concatenated string (*string1*).

The **strchr** function returns a pointer to the first occurrence of *c* in *string*. The character *c* may be the null character (**'\0'**); the terminating null character of *string* is included in the search. The function returns **NULL** if the character is not found.

The **strcmp** function compares *string1* and *string2* lexicographically and returns a value indicating their relationship, as follows:

Value	**Meaning**
< 0	*string1* less than *string2*
$= 0$	*string1* identical to *string2*
> 0	*string1* greater than *string2*

The **strcmpi** and **stricmp** functions are case-insensitive versions of **strcmp**. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison, so *string1* and *string2* are compared without regard to case.

The **strcpy** function copies *string2*, including the terminating null character, to the location specified by *string1*, and returns *string1*.

The **strcspn** function returns the index of the first character in *string1* that belongs to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters not in *string2*. Terminating null characters are not considered in the search. If *string1* begins with a character from *string2*, **strcspn** returns 0.

The **strdup** function allocates storage space (with a call to **malloc**) for a copy of *string* and returns a pointer to the storage space containing the copied string. The function returns **NULL** if storage cannot be allocated.

Note

> The **strcmpi**, **stricmp**, and **strdup** functions are not part of the ANSI definition but are instead Microsoft extensions to it. They should not be used where ANSI portability is desired.

Return Value

The return values for these functions are described above.

See Also

strncat, **strncmp**, **strncpy**, **strnicmp**, **strrchr**, **strspn**

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "XYZabbc This is a string!";
char template[100] = "XYZabbc This is A STRING!";
char *newstring;
char *result;
int numresult;

main()
 {

  /* Construct computer program
   ** using "strcpy" and "strcat"
   */
  strcpy(string, "computer");
  result = strcat(string, " program");
  printf("Result = %s\n", result);

  /* Find the first occurrence of 'a': */
  result = strchr(string,'a');
  printf("String after an \"a\" is %s\n", result);

  /* Compare one string against another  */
  /*  and report whether less than, greater than */
  /*  or equal to: */
  numresult = strcmp(string,template);
  printf( "\"%s\" is %s \"%s\"\n", string, numresult ?
        ( numresult > 0 ? "greater than" : "less than" ) :
        "equal to", template );

  /* Compare string with regard to case */
  numresult = strcmpi("hello", "HELLO");
  printf("\"%s\" is %s \"%s\"\n", "hello", numresult ?
        ( numresult > 0 ? "greater than" : "less than" ) :
        "equal to", "HELLO" );
 /* Make a copy of a string */
 printf("\"%s\"  \"%s\\"\n", template, string);
 result = strcpy(template,string);
 printf("\"%s\"  \"%s\"\n", template, string);

 /* Search for a's, b's, or c's */
 strcpy(string, "xyzabc" );
 numresult = strcspn(string,"abc");
 printf( "The location of the first a, b, or c is %d\n",
       numresult );

 /* Make newstring point to a duplicate of string: */
 newstring = strdup(string);
 printf("The new string is %s\n", newstring);
}
```

This program demonstrates the uses of the **strcat**, **strchr**, **strcmp**, **strcmpi**, **strcpy**, **strcspn**, and **strdup** functions.

_strdate

Summary

#include <time.h>

char *_strdate(*date***);**
char ***date*;** Current date

Description

The **_strdate** function copies the date to the buffer that *date* points to, formatted

```
mm/dd/yy
```

where `mm` is two digits representing the month, `dd` is two digits representing the day of the month, and `yy` is the last two digits of the year. For example, the string

```
12/05/88
```

represents December 5, 1988.

The buffer must be at least nine bytes long.

Return Value

There is no error return.

See Also

asctime, **ctime**, **gmtime**, **localtime**, **mktime**, **time**, **tzset**

Example

```
#include <time.h>

main()
{
 char buffer [9];

 _strdate(buffer);
 printf("The current date is %s \n", buffer);
}
```

This program prints the current date.

strerror, _strerror

Summary

#include <string.h>	Required only for function declarations
char *strerror(*errnum***);**	ANSI version
int *errnum***;**	Error number
char *_strerror(*string***);**	Non-ANSI version
char ****string*;**	User-supplied message
int errno;	Error number
int sys_nerr;	Number of system messages
char *sys_errlist[sys_nerr];	Array of error messages

Description

The **strerror** function maps *errnum* to an error-message string, returning a pointer to the string. The function itself does not actually print the message; for that, you need to call an output function such as **printf**.

If *string* is passed as **NULL**, **_strerror** returns a pointer to a string containing the system error message for the last library call that produced an error. The error-message string is terminated by the new-line character ('\n').

If *string* is not equal to **NULL**, then **_strerror** returns a pointer to a string containing, in order, your string message, a colon, a space, the system error message for the last library call producing an error, and a new-line character. Your string message can be a maximum of 94 bytes long.

Unlike **perror**, **_strerror** alone does not print any messages. To print the message returned by **_strerror** to **stderr**, your program will need a **printf** statement, as shown in the following lines:

```
if ((access("datafile",2)) == -1)
     printf(_strerror(NULL));
```

The actual error number for **_strerror** is stored in the variable **errno**, which should be declared at the external level. The system error messages are accessed through the variable **sys_errlist**, which is an array of messages ordered by error number. The **_strerror** function accesses the appropriate error message by using the **errno** value as an index to **sys_errlist**. The value of the variable **sys_nerr** is defined as the maximum number of elements in the **sys_errlist** array.

To produce accurate results, **_strerror** should be called immediately after a library routine returns with an error. Otherwise, the **errno** value may be overwritten by subsequent calls.

Note

> The **_strerror** function under Microsoft C Version 5.0 is identical to the Version 4.0 **strerror** function. The name was altered to permit the inclusion in Microsoft C Version 5.0 of the ANSI-conforming **strerror** function. The **_strerror** function is not part of the ANSI definition, but is instead a Microsoft extension to it, and should not be used where portability is desired. For ANSI compatibility, use **strerror** instead.

■ Return Value

The **strerror** function returns a pointer to the error-message string. The string can be overwritten by subsequent calls to **strerror**.

The **_strerror** function returns no value.

■ See Also

clearerr, **ferror**, **perror**

Note

> Under MS-DOS, some of the **errno** values listed in **errno.h** are not used. See Appendix A, "Error Messages," for a list of **errno** values used on MS-DOS, and the corresponding error messages. The **_strerror** function prints an empty string for any **errno** value not used under MS-DOS.

Example

```
#include <string.h>
#include <errno.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

extern int errno;
int errnum;
int fh1, fh2;

main()
{
 errnum=0;
 if ((fh1=open("xxxx",O_RDONLY)) == -1)
  errnum=errno;
 fh2=open("yyyy",O_RDONLY);
 /* Other code that may set the errno value.*/
 if (errnum != 0)
  printf(strerror(errnum));
}
```

The program shown above tries to open files xxxx and yyyy. If an error occurs opening xxxx, the variable errnum is set to the **errno** value returned by **open**. Other code that may alter the **errno** value is then executed. Later, the saved **errno** value in errnum is checked and, if nonzero, an error message assigned to it by **strerror** is printed. If file xxxx does not exist, the example will print the following message:

```
No such file or directory
```

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

main()
 {                         /* Since "xxxx" does not exist, */
                           /* both open attempts will fail  */
nt fh1, fh2;
  fh1 = open("xxxx", O_RDONLY);
  if (fh1 == -1);
     printf(_strerror("Open failed on input file "));

  fh2 = open("xxxx", O_WRONLY¦O_TRUNC, S_IREAD¦S_IWRITE);
  if (fh2 == -1)
    printf(_strerror("Open failed on output file "));
 }
```

This program tries to open files for input and output. If an error occurs, the program uses **_strerror** to tag an error message onto the front of the standard error message and then displays the entire error message.

strlen

Summary

#include <string.h>	Required only for function declarations
size_t strlen(*string*);	
char **string*;	Null-terminated string

Description

The **strlen** function returns the length in bytes of *string*, not including the terminating null character (**'\0'**).

Return Value

The **strlen** function returns the string length. There is no error return.

Example

```
#include <string.h>
#include <stdio.h>

char *string = "some space";
size_t result;

main()
 {
  result = strlen(string);             /* result is 10 */
  printf("The size of the string is %d", result);
 }
```

This program uses **strlen** to determine the length of the string named `string`.

Summary

#include <string.h>	Required only for function declarations
char *strlwr(*string***);**	
char ****string*;**	String to be converted

Description

The **strlwr** function converts any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

Return Value

The **strlwr** function returns a pointer to the converted string. There is no error return.

See Also

strupr

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "This Was a Mixed-Case String", *copy;

main()
 {
  copy = strlwr(strdup(string));
  printf("The result string is: %s",copy);
}
```

This program duplicates a string named `string`, then uses **strlwr** to convert all uppercase letters in the copy to lowercase.

Summary

`#include <string.h>`	Required only for function declarations
`char *strncat(string1, string2, n);`	Appends *n* characters of *string2* to *string1*
`char *string1;`	Destination string
`const char *string2;`	Source string
`size_t n;`	Number of characters appended
`int strncmp(string1, string2, n);`	Compares first *n* characters of strings
`const char *string1;`	
`const char *string2;`	
`size_t n;`	Number of characters compared
`int strnicmp(string1, string2, n);`	Compares first *n* characters of strings without regard to case
`const char *string1;`	
`const char *string2;`	
`size_t n;`	Number of characters compared
`char *strncpy(string1, string2, n);`	Copies *n* characters of *string2* to *string1*
`char *string1;`	Destination string
`const char *string2;`	Source string
`size_t n;`	Number of characters copied
`char *strnset(string, c, n);`	Initializes first *n* characters of *string*
`char *string;`	String to be initialized
`int c;`	Character setting
`size_t n;`	Number of characters set

Description

The **strncat**, **strncmp**, **strnicmp**, **strncpy**, and **strnset** functions operate on, at most, the first *n* characters of null-terminated strings.

The **strncat** function appends, at most, the first *n* characters of *string2* to *string1*, terminates the resulting string with a null character (**'\0'**), and returns a pointer to the concatenated string (*string1*). If *n* is greater than the length of *string2*, the length of *string2* is used in place of *n*.

The **strncmp** function compares, at most, the first *n* characters of *string1* and *string2* lexicographically and returns a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	*substring1* less than *substring2*
= 0	*substring1* equivalent to *substring2*
> 0	*substring1* greater than *substring2*

The **strnicmp** function is a case-insensitive version of **strncmp**: **strnicmp** converts all alphabetic characters in the two strings *string1* and *string2* to lowercase before comparing them. This action results in all uppercase and lowercase forms of a letter being considered equivalent.

The **strncpy** function copies exactly *n* characters of *string2* to *string1* and returns *string1*. If *n* is less than the length of *string2*, a null character (**'\0'**) is *not* appended automatically to the copied string. If *n* is greater than the length of *string2*, the *string1* result is padded with null characters (**'\0'**) up to length *n*.

Note

The behavior of **strncpy** is undefined if the address ranges of *string1* and *string2* overlap.

The **strnset** function sets, at most, the first *n* characters of *string* to the character *c* and returns a pointer to the altered string. If *n* is greater than the length of *string*, the length of *string* is used in place of *n*.

Note

The **strnicmp** and **strnset** functions are not part of the ANSI definition but are instead Microsoft extensions to it, and should not be used where ANSI portability is desired.

See Also

strcat, **strcmp**, **strcpy**, **strset**

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "XYZabbc This is a string!";
char copy[100] = "This is a different string";
char *result;
char suffix[100] = " this is even more string..";
int numresult;

main()
 {

/* Combine strings with no more than */
/* 100 characters of suffix:         */
  printf("String before = %s\n", string);
  result = strncat(string,suffix,100);
  printf("String after = %s\n", string);

  /* Determine ordering of two strings */
  /*  but only consider first 7 chars:  */
  strcpy(string, "programming");
  numresult = strncmp(string,"program",7);
  printf("\"%s\" is %s \"%s\"\n", string,
  numresult ? (numresult > 0 ?
          "greater than" : "less than") : "equal to",
          "program");

  /* Copy at most 99 chars of "string" */
  printf("%s \"%s\"\n", copy, string);
  result = strncpy(copy,string,99);
  copy[99] = '\0';          /* Null terminate the result */
  printf("%s  %s \n", copy, string);

  /* Set not more than 4 characters of a */
  /*  string to be x's:                  */
  result = strnset("computer",'x',4);
  printf( "%s\n", result );   /* Result is now "xxxxuter". */
 }
```

This program demonstrates the uses of the **strncat**, **strncmp**, **strnicmp**, and **strnset** functions.

Summary

#include <string.h>	Required only for function declarations
char *strpbrk(*string1*, *string2***);**	Finds any character from *string2* in *string1*
const char ****string1*;**	Source string
const char ****string2*;**	Character set

Description

The **strpbrk** function finds the first occurrence in *string1* of any character from *string2*. The terminating null character (**'\0'**) is not included in the search.

Return Value

The **strpbrk** function returns a pointer to the first occurrence of any character from *string2* in *string1*. A null pointer indicates that *string1* and *string2* have no characters in common.

See Also

strchr, **strrchr**

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "Find an 'a' or 'b' in this string",*result;

main()
 {                        /* Return pointer to first */
                          /* 'a' or 'b' in "string"  */
  result = strpbrk(string, "ab");
  printf("The remainder of the string
starting at the first\n");
  printf("'a' or 'b' is:  %s",result);
 }
```

This program uses **strpbrk** to find the first occurrence of `a` or `b` in the string named `string`.

strrchr

Summary

#include <string.h>	Required only for function declarations
char *strrchr(*string, c***);**	Finds last occurrence of *c* in *string*
const char ****string*;**	Searched string
int *c***;**	Character to be located

Description

The **strrchr** function finds the last occurrence of the character *c* in *string*. The string's terminating null character ('**\0**') is included in the search. (Use **strchr** to find the first occurrence of *c* in *string*.)

Return Value

The **strrchr** function returns a pointer to the last occurrence of *c* in *string*. A null pointer is returned if the given character is not found.

See Also

strchr, **strpbrk**

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "Find the last 'a' in this string", *result;

main()
 {

  /* Return a pointer to the last: 'a' */
  result = strrchr(string, 'a');
  printf("The remainder of the string starting at the
          first\n");
  printf("'a' is:  %s",result);
 }
```

This program uses **strrchr** to find the last occurrence of `a` in the string named `string`.

Summary

#include <string.h>	Required only for function declarations
char *strrev(*string*);	
char ****string*;**	String to be reversed

Description

The **strrev** function reverses the order of the characters in *string*. The terminating null character ('**0**') remains in place.

Return Value

The **strrev** function returns a pointer to the altered string. There is no error return.

See Also

strcpy, **strset**

Example

```
#include <string.h>
#include <stdio.h>

char string[100];
int result;

main()
 {
  printf("Input a string and I will tell \
  you if it is a palindrome: ");
  gets(string);
  /* Reverse string and compare: */
  result = strcmp(string,strrev(strdup(string)));

  if (result == 0)
    printf("The string \"%s\" is a palindrome\n\n",
        string);
  else
    printf("The string \"%s\" is not a palindrome\n\n",
        string);
 }
```

This program checks an input string to see whether it is a palindrome: that is, whether it reads the same forward and backward. The program checks this by comparing a string named `string` with a copy of `string` that has been reversed using **strrev**.

Summary

#include <string.h>	Required only for function declarations
char *strset(*string*, *c***);**	
char ****string*;**	String to be set
int *c***;**	Character setting

Description

The **strset** function sets all characters of *string* to *c*, except the terminating null character (**'\0'**).

Return Value

The **strset** function returns a pointer to the altered string. There is no error return.

See Also

strnset

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "Fill the string with something" ,
                    *result;

main()
 {
  printf("The string before 'strset' is used: \"%s\"\n",
          string);
  result = strset(string,' ');
  printf("The string after 'strset' was used: \"%s\"\n",
          result);
 }
```

This program uses **strset** to fill the string named `string` with blanks.

strspn

Summary

#include <string.h>	Required only for function declarations
size_t strspn(*string1*, *string2*);	
const char ****string1*;**	Searched string
const char ****string2*;**	Character set

Description

The **strspn** function returns the index of the first character in *string1* that *does not* belong to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters from *string2*. The null character (**'\0'**) terminating *string2* is not considered in the matching process. If *string1* begins with a character not in *string2*, **strspn** returns 0.

Return Value

The **strspn** function returns an integer value specifying the length of the segment in *string1* consisting entirely of characters in *string2*.

See Also

strcspn

Example

```
#include <string.h>
#include <stdio.h>

char *string = "cabbage";
int result;

main()
 {
  result = strspn(string, "abc");      /* result = 5 */
  printf("The string starting with \"abc\" is %d bytes long.",
          result);
 }
```

This program uses **strspn** to determine the length of the segment in the string `cabbage` consisting of `a`'s, `b`'s, and `c`'s.

Summary

#include <string.h>	Required only for function declarations
char *strstr(*string1, string2***);**	
const char ****string1*;**	Searched string
const char ****string2*;**	String to search for

Description

The **strstr** function returns a pointer to the first occurrence of *string2* in *string1.*

Return Value

The **strstr** function returns either a pointer to the first occurrence of *string2* in *string1,* or **NULL** if it does not find *string2* in *string1.*

See Also

strcspn

Example

```
#include <string.h>
#include <stdio.h>

main()
 {
  char *string1 = "needle in a haystack";
  char *string2 = "hay";

  printf("%s\n", strstr(string1,string2));
 }
```

Output:

```
haystack
```

This program uses **strstr** to return a pointer to the first location of `hay` in the string `string1`, then prints the remainder of the string.

_strtime

Summary

#include <time.h>

char *_strtime(*time*);
**char ****time*; Time string

Description

The **_strtime** function copies the current time into the buffer that *time* points to, formatted

```
hh:mm:ss
```

where `hh` is two digits representing the hour in 24-hour notation, `mm` is two digits representing the minutes past the hour, and `ss` is two digits representing seconds. For example, the string

```
18:23:44
```

represents 23 minutes and 44 seconds past 6 PM.

The buffer must be at least nine bytes long.

Return Value

There is no error return.

See Also

asctime, **ctime**, **gmtime**, **localtime**, **mktime**, **time**, **tzset**

Example

```
#include <time.h>

main()
{
 char buffer [9];

 _strtime(buffer);
 printf("The current time is %s \n", buffer);
}
```

This program prints the current time.

strtod, strtol, strtoul

Summary

#include <stdlib.h>

double strtod(*nptr*, *endptr***);**	Converts string to **double**
const char **nptr*;	String to convert
char ***endptr*;	End of scan
long strtol(*nptr*, *endptr*, *base***);**	Converts string to **long** decimal integer
const char **nptr*;	String to convert
char ***endptr*;	End of scan
int *base*;	Number base to use
unsigned long int strtoul(*nptr*, *endptr*, *base***);**	Converts string to **unsigned long** decimal
const char **nptr*;	String to convert
char ***endptr*;	End of scan
int *base*;	Number base to use

Description

The **strtod**, **strtol**, and **strtoul** functions convert a character string to a double-precision value, a long-integer value, or an unsigned-long-integer value, respectively. The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. These functions stop reading the string at the first character they cannot recognize as part of a number. This character may be the null (**'\0'**) at the end of the string. With **strtol** or **strtoul** this terminating character can also be the first numeric character greater than or equal to *base*. If *endptr* is not the null character, it points to the character that stopped the scan.

The **strtod** function expects *nptr* to point to a string with the following form:

⟦*whitespace*⟧⟦*sign*⟧⟦*digits*⟧⟦*.digits*⟧⟦ {**d** | **D** | **e** | **E**} ⟦*sign*⟧*digits*⟧

S

The first character that doesn't fit this form stops the scan.

The **strtol** function expects *nptr* to point to a string with the following form:

⟦*whitespace*⟧ ⟦*sign*⟧ ⟦**0**⟧ ⟦{ **x** | **X** }⟧ ⟦*digits*⟧

The **strtoul** function expects *nptr* to point to a string having this form:

⟦*whitespace*⟧ ⟦**0**⟧ ⟦{ **x** | **X** }⟧ ⟦*digits*⟧

If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base: if the first character is 0 and the second character is a digit '0' – '7', then the string is interpreted as an octal integer; if the first character is '0' and the second character is 'x' or 'X', then the string is interpreted as a hexadecimal integer; if the first character is '1' – '9', then the string is interpreted as a decimal integer. The letters from 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 – 35; only letters whose assigned values are less than *base* are permitted.

■ Return Value

The **strtod** function returns the value of the floating-point number, except when the representation would cause an overflow, in which case it returns ±**HUGE_VAL**. The function returns 0 if no conversion could be performed or an underflow occurred.

The **strtol** function returns the value represented in the string, except when the representation would cause an overflow, in which case it returns **LONG_MAX** or **LONG_MIN**. The functions returns 0 if no conversion could be performed.

The **strtoul** function returns the converted value, if any. If no conversion can be performed, the function returns 0. The function returns **ULONG_MAX** on overflow.

In all three functions **errno** is set to **ERANGE** if overflow or underflow occurs.

■ See Also

atof, **atol**

Example

```
#include <stdlib.h>
#include <stdio.h>

main()
 {
  char * string, *stopstring;
  double x;
  long l;
  unsigned long ul;
  int bs;

  string = "3.1415926This stopped it";
  x = strtod(string,&stopstring); /* Convert the string */
  printf("string = %s\n",string);
  printf("   strtod = %f\n",x);
  printf("   Stopped scan at %s\n\n", stopstring);

  string = "-10110134932";
  printf("string = %s\n",string);
  /* Convert string using base 2, 4, & 8: */
  for (bs = 2; bs <= 8; bs *= 2)
   {
    /* Convert the string: */
    l = strtol(string,&stopstring,bs);
    printf("   strtol = %ld (base %d)\n", l, bs);
    printf("   Stopped scan at %s\n\n", stopstring);
   }

  string = "10110134932";
  printf("string = %s\n",string);
  /* Convert string using base 2, 4, & 8: */
  for (bs = 2; bs <= 8; bs *= 2)
   { /* Convert the string: */
    ul = strtoul(string,&stopstring,bs);
    printf("   strtol = %ld (base %d)\n", ul, bs);
    printf("   Stopped scan at %s\n\n", stopstring);
   }
}
```

Output:

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at This stopped it

string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at 34932

   strtol = 4423 (base 4)
   Stopped scan at 4932

   strtol = 2134108 (base 8)
   Stopped scan at 932
```

Strings are converted to numbers using the `strtod` and `strtol` functions. This program uses **strtod** to convert a string to a double-precision value; **strtol** to convert a string to an integer value; and **strtoul** to convert a string to three long-integer values.

strtok

Summary

#include <string.h>	Required only for function declarations
char *strtok(*string1*, *string2*);	Finds token in *string1*
**char ****string1*;	String containing token(s)
**const char ****string2*;	Set of delimiter characters

Description

The **strtok** function reads *string1* as a series of zero or more tokens and *string2* as the set of characters serving as delimiters of the tokens in *string1*. The tokens in *string1* may be separated by one or more of the delimiters from *string2*. The tokens are broken out of *string1* by a series of calls to **strtok.**

In the first call to **strtok** for *string1*, **strtok** searches for the first token in *string1*, skipping leading delimiters. A pointer to the first token is returned.

To read the next token from *string1*, call **strtok** with a **NULL** value for the *string1* argument. The **NULL** *string1* argument causes **strtok** to search for the next token in the previous token string. The set of delimiters may vary from call to call, so *string2* can take any value.

Note

> Calls to **strtok** will modify *string1*, since each time **strtok** is called it inserts a **NULL** character (**'\0'**) after the token in *string1*.

Return Value

The first time **strtok** is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, **strtok** returns a pointer to the next token in the string. A null pointer is returned when there are no more tokens. All tokens are null terminated.

See Also

strcspn, **strspn**

Example

```
#include <string.h>
#include <stdio.h>

char *string = "a string,of ,,tokens ";
char *token;

main()
 {
  /* Establish string and get the first token: */
  token = strtok(string," ,");
  while (token != NULL)
  /* While there are tokens in "string" */
   {
    printf("The token is:  %s\n", token);

    /* Get next token: */
    token = strtok(NULL," ,");
   }
 }
```

In this program, a loop uses **strtok** to print all the tokens (separated by commas or blanks) in the string named `string`.

strupr

Summary

#include <string.h>	Required only for function declarations
char *strupr(*string***);**	
char ****string*;**	String to be capitalized

Description

The **strupr** function converts any lowercase letters in *string* to uppercase. Other characters are not affected.

Note

> The **strupr** function is not part of the ANSI definition, but is instead a Microsoft extension to it, and should not be used where ANSI portability is desired.

Return Value

The **strupr** function returns a pointer to the converted string. There is no error return.

See Also

strlwr

Example

```
#include <string.h>
#include <stdio.h>

char string[100] = "This Was a Mixed-Case String", *copy;

main()
 {
  copy = strupr(strdup(string));
  printf("The result string is: %s",copy);
}
```

This program duplicates a string named `string` and uses **strupr** to convert all lowercase letters in the copy to uppercase.

swab

Summary

#include <stdlib.h>	Required only for function declarations
void swab(*source, destination, n***);**	
char ****source*;**	Data to be copied and swapped
char ****destination*;**	Storage location for swapped data
int *n***;**	Number of bytes copied

Description

The **swab** function copies *n* bytes from *source*, swaps each pair of adjacent bytes, and stores the result at *destination*. The integer *n* should be an even number to allow for swapping. The **swab** function is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Return Value

There is no return value.

See Also

fgetc, **fputc**

Example

```
#define NBYTES 18

char from[NBYTES], to[NBYTES];

main()
 {
  strcpy( from, "badcfehgjilknmporq");
  strcpy( to, "..................");
  printf( "%s  %s\n", from, to );

  swab(from,to,NBYTES);    /* to = "abcdefghijklmnopqr" */

  printf( "%s  %s\n", from, to );
 }
```

This program uses **swab** to copy the string named `from` to the string named `to` and swap each adjacent pair of bytes.

Summary

# include <stdlib.h>	For ANSI compatibility
# include <process.h>	For UNIX System V compatibility Include file required only for function declarations
int system(*string*);	
**const char ****string*;	Command to be executed

Description

The **system** function passes *string* to the command interpreter and executes the string as a MS-DOS command. The **system** function refers to the **COMSPEC** and **PATH** environment variables to locate the MS-DOS file **COMMAND.COM**, which is used to execute the *string* command.

If *string* is **NULL**, the function merely looks to see whether **COMMAND.COM** is present.

C 4.0 Difference

Under Microsoft C, Version 4.0, **system** does not allow a null value for *string*; it also returns an error.

Return Value

If *string* is not **NULL**, the function returns the value 0 if *string* is successfully executed. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
E2BIG	The argument list for the command exceeds 128 bytes, or the space required for the environment information exceeds 32K.
ENOENT	**COMMAND.COM** cannot be found.
ENOEXEC	The **COMMAND.COM** file has an invalid format and is not executable.

ENOMEM	Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

If *string* is **NULL** and if it finds **COMMAND.COM**, the function returns a nonzero value. If it does not find **COMMAND.COM**, it returns 0 and sets **errno** to ENOENT.

See Also

exec functions, **exit**, **_exit**, **spawn** functions

Example

```
#include <stdlib.h>

int result;

main()
{
 /* Place version number in "result.log": */
 result = system("ver >>result.log");
 /* Type "result.log" to the screen:     */
 result = system("type result.log");
}
```

This program uses **system** to place the MS-DOS version number in a file named `result.log` and then displays `result.log` on the screen.

Summary

#include <math.h>

double tan(*x*);	Calculates tangent of *x*
double tanh(*x*);	Calculates hyperbolic tangent of *x*
double *x*;	Angle in radians

Description

The **tan** and **tanh** functions return the tangent and hyperbolic tangent of *x*, respectively.

Return Value

The **tan** function returns the tangent of *x*. If *x* is large, a partial loss of significance in the result may occur, so **tan** sets **errno** to **ERANGE** and generates a **PLOSS** error. If *x* is so large that significance is totally lost, **tan** prints a **TLOSS** error message to **stderr**, sets **errno** to **ERANGE**, and returns 0.

There is no error return for **tanh**.

See Also

acos, **asin**, **atan**, **atan2**, **cos**, **cosh**, **sin**, **sinh**

Example

```
#include<math.h>
#include<stdio.h>

main()
 {
  double pi = 3.1415926535;
  double x = tan(pi/4);                    /* x is 1.0 */
  double y = tanh(x);                 /* y is 0.761594 */
  printf("The tan(%f) = %f\n",pi/4,x);
  printf("The tanh(%f) = %f\n",x,y);
 }
```

This program displays the tangent of $\pi/4$ and hyperbolic tangent of 1.0.

tell

Summary

#include <io.h>	Required only for function declarations
long tell(*handle*);	
int *handle*;	Handle referring to open file

Description

The **tell** function gets the current position of the file pointer (if any) associated with *handle*. The position is expressed as the number of bytes from the beginning of the file.

Return Value

A return value of –1L indicates an error, and **errno** is set to **EBADF** to indicate an invalid file-handle argument. On devices incapable of seeking, the return value is undefined.

See Also

ftell, **lseek**

Example

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

int fh;
long position;
main()
{
 fh = open("data",O_RDONLY);
 position = tell(fh);                  /* Position = 0    */
 printf("position = %ld\n", position);
 lseek(fh, -3L, SEEK_END);
 position = tell(fh);      /* Position = file length -3 */
 printf("position = %ld\n", position);
 lseek(fh, position, SEEK_SET); /* Put pointer back to */
}                               /* previous position   */
```

This program uses **tell** to find the beginning and a position three bytes from the end of the file named `data`.

Summary

#include <stdio.h>

char *tmpnam(*string*);	Creates temporary file in directory defined by **P_tmpdir**
**char ****string*;	Pointer to temporary name
char *tempnam(*dir, prefix*);	Creates temporary file in another directory
**char ****dir*;	Target directory if **TMP** not defined
**char ****prefix*;	File-name prefix

Description

The **tmpnam** function generates a temporary file name that can be used as a temporary file. This name is stored in *string.* If *string* is NULL, then **tmpnam** leaves the result in an internal static buffer. Thus, any subsequent calls will destroy this value. If *string* is not NULL, it is assumed to point to an array of at least **L_tmpnam** bytes, where the value of **L_tmpnam** is defined in the **stdio.h** include file. The function will generate unique file names for up to **TMP_MAX** calls.

The character string that **tmpnam** creates consists of the path prefix defined by the **P_tmpdir** entry in **stdio.h**, followed by a sequence consisting of the digit characters '0' through '9'; the numerical value of this string can range from 1 to 65,535. Changing the definitions of **L_tmpnam** or **P_tmpdir** in **stdio.h** does not change the operation of **tmpnam**.

The **tempnam** function allows the user to create a temporary file in another directory. The *prefix* is the prefix to the file name. The **tempnam** function uses **malloc** to allocate space for the file name; the user is responsible for freeing this space when it is no longer needed. The **tempnam** function looks for the file with the given name in the following directories, listed in order of precedence:

Directory Used	**Conditions**
Directory specified by **TMP**	**TMP** environment variable is set, and directory specified by **TMP** exists.
dir argument to **tempnam**	**TMP** environment variable is not set, or directory specified by **TMP** does not exist.

P_tmpdir in **stdio.h**	The *dir* argument is **NULL**, or *dir* is name of nonexistent directory.
Current working directory	**P_tmpdir** does not exist.

If all this fails, **tempnam** returns the value **NULL**.

Return Value

The **tmpnam** and **tempnam** functions both return a pointer to the name generated, unless it is impossible to create this name, or the name is not unique. If the name cannot be created or if it already exists, **tmpnam** and **tempnam** return the value **NULL**.

See Also

tmpfile

Example

```
#include <stdio.h>

main()
 {
  char *name1, *name2;

  /* Create a temporary file name for */
  /*  the current working directory:   */
  if ((name1 = tmpnam(NULL)) != NULL)
    printf("%s is safe to use as a temporary file.\n", name1);
  else
    printf("Cannot create a unique file name\n");

  /* Create a temporary file name for    */
  /*  directory "c:\tmp" with the prefix "stq": */
  if ((name2 = tempnam("c:\tmp","stq")) != NULL)
    printf("%s is safe to use as a temporary file.\n", name2);
  else
    printf("Cannot create a unique file name\n");
 }
```

This program uses **tempnam** to create a file name that is unique to the current working directory, then uses **tempnam** to create a file name that is unique in `c:\tmp` with a prefix of `stq`. This behavior assumes that the **TMP** environment variable is not set.

Summary

#include <time.h>	Required only for function declarations
time_t time(*timeptr*);	
time_t ****timeptr*;	Storage location for time

Description

The **time** function returns the number of seconds elapsed since 00:00:00 Greenwich mean time (GMT), January 1, 1970, according to the system clock. The system time is first adjusted according to the **_timezone** system variable, which is explained in the **_tzset** reference page.

The return value is stored in the location given by *timeptr*. This parameter may be **NULL**, in which case the return value is not stored.

Return Value

The **time** function returns the time in elapsed seconds. There is no error return.

See Also

asctime, **ftime**, **gmtime**, **localtime**, **tzset**, **utime**

Example

```
#include <time.h>
#include <stdio.h>

time_t ltime;

main()
 {
  time(&ltime);
  printf("The time is %s\n",ctime(&ltime));
 }
```

This program uses **time** to obtain the current time in **time_t** format, then displays this time.

Summary

#include <stdio.h>

FILE *tmpfile(void);	Pointer to **FILE** structure

Description

The **tmpfile** function creates a temporary file and returns a pointer to that file. If the file cannot be opened, **tmpfile** returns a null pointer.

This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **rmtmp** is called, assuming that the current working directory does not change. The temporary file is opened in **w+b** (binary read/write) mode.

C 4.0 Difference

> The C 4.0 version of **tmpfile** opens the temporary file in the **w+** mode, and the translation mode is set by the default mode variable **_fmode**.

Return value

If successful, the **tmpfile** function returns a stream pointer. Otherwise, it returns a null pointer.

See Also

rmtmp, **tempnam**, **tmpnam**

Example

```
#include <stdio.h>

FILE *stream;
char tempstring[] = "String to be temporarily written";

main()
{
 if ((stream = tmpfile()) == NULL) /* Create temporary file */
  perror("Could not open new temporary file");
 else {
  fprintf(stream, "%s", tempstring);
  printf("Temporary file was created, "
         "and \"tempstring\" was output");
 }
 rmtmp();              /* Remove temporary file */
}
```

This program uses **tmpfile** to create a temporary file, then deletes this file.

toascii – _toupper

Summary

#include <ctype.h>

int toascii(*c*);	Converts *c* to ASCII character
int tolower(*c*);	Converts *c* to lowercase if appropriate
int _tolower(*c*);	Converts *c* to lowercase
int toupper(*c*);	Converts *c* to uppercase if appropriate
int _toupper(*c*);	Converts *c* to uppercase
int *c*;	Character to be converted

Description

The **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper** macros convert a single character as specified.

The **toascii** macro sets all but the low-order 7 bits of *c* to 0, so that the converted value represents a character in the ASCII character set. If *c* already represents an ASCII character, *c* is unchanged.

The **tolower** macro converts *c* to lowercase if *c* represents an uppercase letter. Otherwise, *c* is unchanged. The **_tolower** macro is a version of **tolower** to be used only when *c* is known to be uppercase. The result of **_tolower** is undefined if *c* is not an uppercase letter.

The **toupper** macro converts *c* to uppercase if *c* represents a lowercase letter. Otherwise, *c* is unchanged. The **_toupper** macro is a version of **toupper** to be used only when *c* is known to be lowercase. The result of **_toupper** is undefined if *c* is not a lowercase letter.

Note

The **toascii**, **_tolower**, and **_toupper** routines are not part of the ANSI definition, but are instead Microsoft extensions to it, and should not be used where ANSI portability is desired.

Return Value

The **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper** macros return the possibly converted character *c*. There is no error return.

See Also

isalnum, **isalpha**, **isascii**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**

Note

> These routines are implemented as macros. However, **tolower** and **toupper** are also implemented as functions because the macro versions do not correctly handle arguments with side effects. The function versions can be used by removing the macro definitions through #**undef** directives or by not including **ctype.h**. Function declarations of **tolower** and **toupper** are given in **stdlib.h**.

Example

```
#include <stdio.h>
#include <ctype.h>

int ch;

main()
{
 for ( ch = 0; ch <= 0x7f; ch++ ) {
  printf(" toupper =%#04x", toupper(ch));
  printf(" tolower =%#04x", tolower(ch));
  if (islower(ch) )
   printf(" _toupper =%#04x", _toupper(ch));
  if (isupper(ch))
   printf(" _tolower =%#04x", _tolower(ch));
  putchar('\n');
 }
}
```

This program uses **toupper** and **tolower** to analyze all characters between 0x0 and 0x7F. It also applies **_toupper** and **_tolower** to any code in this range for which these functions make sense.

tzset

Summary

#include <time.h>	Required only for function declarations
void tzset(void);	
int daylight;	Daylight-saving-time flag
long timezone;	Difference in seconds from GMT
char *tzname[2]	Three-letter time-zone strings

Description

The **tzset** function uses the current setting of the environment variable **TZ** to assign values to three global variables: **daylight**, **timezone**, and **tzname**. These variables are used by the **ftime** and **localtime** functions to make corrections from Greenwich Mean Time (GMT) to local time, and by **time** to compute GMT from system time.

The value of the environment variable **TZ** must be a three-letter time-zone name, such as PST, followed by an optionally signed number giving the difference in hours between GMT and local time. The number may be followed by a three-letter daylight-saving-time (DST) zone, such as PDT. For example, "PST8PDT" represents a valid **TZ** value for the Pacific time zone. If DST is never in effect, as is the case in certain states and localities, **TZ** should be set without a DST zone.

The following values are assigned to the variables **daylight**, **timezone**, and **tzname** when **tzset** is called:

Variable	Value
timezone	The difference in seconds between GMT and local time
daylight	Nonzero value if a daylight-saving-time zone is specified in the **TZ** setting; otherwise, 0
tzname[0]	The string value of the three-letter time-zone name from the **TZ** setting
tzname[1]	The string value of the daylight-saving-time zone, or an empty string if the daylight-saving-time zone is omitted from the **TZ** setting

If **TZ** is not currently set, the default is PST8PDT, which corresponds to the Pacific time zone. The default for **daylight** is 1; for **timezone**, 28800; for **tzname[0]**, PST; and for **tzname[1]**, PDT.

If the DST zone is omitted from the **TZ** settings, the **daylight** variable will be 0 and the **ftime**, **gmtime**, and **localtime** functions will return 0 for their DST flags.

Note

The **tzset** function is not part of the ANSI definition, but is instead a Microsoft extension to it, and should not be used where ANSI portability is desired.

Return Value

There is no return value.

See Also

asctime, **ftime**, **gmtime**, **localtime**, **time**

Example

```
#include <time.h>
#include <stdio.h>

int daylight;
long timezone;
char *tzname[];
```

```
main()
 {
  putenv("TZ=EST5");
  tzset();

  /* daylight = 0 */
  printf("daylight = %d\n", daylight);

  /* timezone = 18000 */
  printf("timezone = %ld\n", timezone);

  /* tzname[0] = "EST" */
  printf("tzname[0] = %s\n", tzname[0]);
 }
```

This program first sets up the time zone by placing the variable named TZ=EST5 in the environment table. It then uses **tzset** to set the variables named `daylight`, `timezone`, and `tzname`.

Summary

# include <stdlib.h>	Required only for function declarations
char *ultoa(*value, string, radix***);**	
unsigned long *value***;**	Number to be converted
char ****string*;**	String result
int *radix***;**	Base of *value*

Description

The **ultoa** function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value*; it must be in the range 2–36.

Return Value

The **ultoa** function returns a pointer to *string*. There is no error return.

See Also

itoa, **ltoa**

Example

```
#include <stdlib.h>

int radix = 16;
char buffer[40];
char *p;

main()
 {
  p = ultoa(1344115000L,buffer,radix); /* p = "501d9138" */
  printf("buffer= \"%s\"\n", buffer);
 }
```

This program converts the long integer 1,344,115,000 to a string and displays that string.

Summary

#include <sys\types.h>	
#include <sys\stat.h>	
#include <io.h>	Required only for function declarations
int umask(*pmode***);**	
int *pmode***;**	Default permission setting

Description

The **umask** function sets the file-permission mask of the current process to the mode specified by *pmode*. The file-permission mask is used to modify the permission setting of new files created by **creat**, **open**, or **sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The argument *pmode* is a constant expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in **sys\stat.h**. When both constants are given, they are joined with the bitwise-OR operator (¦). The meaning of the *pmode* argument is as follows:

Value	Meaning
S_IWRITE	Writing not allowed (file is read only)
S_IREAD	Reading not allowed (file is write only)

For example, if the write bit is set in the mask, any new files will be read only.

Note

> Under MS-DOS, all files are readable—it is not possible to give write-only permission. Therefore, setting the read bit with **umask** has no effect on the file's permissions.

Return Value

The **umask** function returns the previous value of *pmode*. There is no error return.

See Also

chmod, **creat**, **mkdir**, **open**

Example

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

int oldmask;

main()
 {

  /* Create read-only files: */
  oldmask = umask(S_IWRITE);
  printf( "oldmask =%#x\n", oldmask );
 }
```

This program uses **umask** to set the file-permission mask so that all future files will be created as read-only files. It also displays the old mask.

ungetc

Summary

#include <stdio.h>

int ungetc(*c***,** *stream***);**	
int *c***;**	Character to be pushed
FILE **stream***;**	Pointer to **FILE** structure

Description

The **ungetc** function pushes the character *c* back onto the input *stream* and clears the end-of-file indicator. The stream must be open for reading. A subsequent read operation on the stream starts with *c*. An attempt to push **EOF** onto the stream using **ungetc** is ignored. The **ungetc** function returns an error value if nothing has yet been read from *stream* or if *c* cannot be pushed back.

Characters placed on the stream by **ungetc** may be erased if **fflush**, **fseek**, **fsetpos**, or **rewind** is called before the character is read from the stream. The file-position indicator will have the same value it had before the characters were pushed back. On a successful **ungetc** call against a text stream, the file-position indicator is unspecified until all the pushed-back characters are read or discarded. On each successful **ungetc** call against a binary stream, the file-position indicator is stepped down; if its value was 0 before a call, the value is undefined after the call.

Return Value

The **ungetc** function returns the character argument *c*. The return value **EOF** indicates a failure to push back the specified character.

See Also

getc, **getchar**, **putc**, **putchar**

Example

```
#include <stdio.h>
#include <ctype.h>

FILE *stream;
int ch;
int result = 0;

main()
 {
  stream = stdin;
  printf("Input an integer: ");

  /* Read in and convert number: */
  while ((ch = getc(stream)) != EOF && isdigit(ch))
    result = result * 10 + ch - '0';

  if (ch != EOF)
    ungetc(ch,stream);     /* Put non-digit back */

  printf("Number = %d\nNext character in stream = \"%c\"\n",
          result, getc(stream));
 }
```

This program first converts a character representation of an unsigned integer to an integer. If the program encounters a character that is not a digit, the program uses **ungetc** to replace it in the stream.

ungetch

Summary

#include <conio.h>	Required only for function declarations
int ungetch(*c*);	
int *c*;	Character to be pushed

Description

The **ungetch** function pushes the character *c* back to the console, causing *c* to be the next character read. The **ungetch** function fails if it is called more than once before the next read. The *c* argument may not be **EOF**.

Return Value

The **ungetch** function returns the character *c* if it is successful. A return value of **EOF** indicates an error.

See Also

cscanf, **getch**, **getche**

Example

```
#include <conio.h>
#include <ctype.h>
#include <stdio.h>

char buffer[100];
int count = 0;
int ch;
```

```
main()
 {
  ch = getche();
  while (isspace(ch))     /* skip preceding white space */
    ch = getche();
  while (count < 99)      /* Gather token */
   {
    if (isspace(ch))      /* End of token */
      break;

    buffer[count++] = ch;
    ch = getche();
   }

  ungetch(ch);             /* Put back delimiter */
  buffer[count] = '\0';   /* Null terminate the token */

  printf( "\ntoken = %s\n", buffer );
 }
```

In this program, tokens are read from the keyboard delimited by blanks and new-line characters. When the program encounters a delimiter, it uses **ungetch** to replace the delimiter in the keyboard buffer.

unlink

Summary

#include <io.h>	Required only for function declarations
#include <stdio.h>	Use either **io.h** or **stdio.h**
int unlink(*path*);	
const char **path*;	Path name of file to be removed

Description

The **unlink** function deletes the file specified by *path*.

Return Value

If successful, **unlink** returns 0; otherwise, it returns –1 and sets **errno** to one of the following constants:

Value	Meaning
EACCES	Path name specifies a directory or a read-only file
ENOENT	File or path name not found

See Also

close, **remove**

Example

```
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

main()
 {
  int result = unlink("tmpfile");
  if (result == -1)
    perror("Couldn't delete tmpfile");
  else
    printf( "Link succesfully removed 'tmpfile'." );
 }
```

This program uses **unlink** to delete a file named `tmpfile`.

Summary

#include <sys\types.h>
#include <sys\utime.h>

int utime(*path*, *times*);	
char ****path*;**	File path name
struct utimbuf ****times*;**	Pointer to stored time values

Description

The **utime** function sets the modification time for the file specified by *path*. The process must have write access to the file; otherwise, the time cannot be changed.

Although the **utimbuf** structure contains a field for access time, under MS-DOS only the modification time is set. If *times* is a null pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type **utimbuf**, defined in **sys\utime.h**. The modification time is set from the **modtime** field in this structure.

Return Value

The **utime** function returns the value 0 if the file-modification time was changed. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EACCES	Path name specifies directory or read-only file
EINVAL	Invalid argument; the *times* argument is invalid
EMFILE	Too many open files (the file must be opened to change its modification time)
ENOENT	File or path name not found

See Also

asctime, **ctime**, **fstat**, **ftime**, **gmtime**, **localtime**, **stat**, **time**

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys\types.h>
#include <sys\utime.h>

main()
 {
  int savestderr;

  if (utime("/tmp/data",NULL) == -1)
    perror("utime failed");
   else
     printf("File time modified.");
 }
```

This program uses **utime** to set the file-modification time to the current time.

Summary

#include <stdarg.h>	Required for ANSI compatibility
#include <varargs.h>	Required for UNIX V compatibility
#include <stdio.h>	
void va_start(*arg-ptr***);**	Macro to set *arg-ptr* to beginning of list of optional arguments (UNIX version only)
void va_start(*arg-ptr, prev-param***);**	Macro to set *arg-ptr* to beginning of list of optional arguments (ANSI version only)
type **va_arg(***arg-ptr, type***);**	Macro to retrieve current argument
void va_end(*arg-ptr***);**	Macro to reset *arg-ptr*
va_list *arg-ptr***;**	Pointer to list of arguments
type	Type of argument to be retrieved
prev-param	Parameter preceding first optional argument (ANSI version only)
va_alist	Name of parameter to called function (UNIX version only)
va_dcl	Declaration of **va_alist** (UNIX version only)

Description

The **va_start**, **va_arg**, and **va_end** macros provide a portable way to access the arguments to a function when the function takes a variable number of arguments. Two versions of the macros are available: the macros defined in **stdarg.h** conform to the proposed ANSI C standard, and the macros defined in **varargs.h** are compatible with the UNIX System V definition.

Both versions of the macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be accessed through the parameter names. The optional arguments are accessed through the **stdarg.h** or **varargs.h** macros, which set a pointer to the first optional argument in the argument list, retrieve arguments from the list, and reset the pointer when argument processing is completed.

The proposed ANSI C standard macros, defined in **stdarg.h**, are used as follows:

1. All required arguments to the function are declared as parameters in the usual way. The **va_dcl** macro is not used with the **stdarg.h** macros.
2. The **va_start** macro sets *arg-ptr* to the first optional argument in the list of arguments passed to the function. The argument *arg-ptr* must have **va_list** type. The argument *prev-param* is the name of the required parameter immediately preceding the first optional argument in the argument list. If *prev-param* is declared with the **register** storage class, the macro's behavior is undefined. The **va_start** macro must be used before **va_arg** is used for the first time.
3. The **va_arg** macro does the following:
   - Retrieves a value of *type* from the location given by *arg-ptr*
   - Increments *arg-ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts

   The **va_arg** macro can be used any number of times within the function to retrieve arguments from the list.
4. After all arguments have been retrieved, **va_end** resets the pointer to **NULL**.

The UNIX System V macros, defined in **varargs.h**, operate in a slightly different manner, as follows:

1. Any required arguments to the function can be declared as parameters in the usual way.
2. The last (or only) parameter to the function represents the list of optional arguments. This parameter must be named **va_alist** (not to be confused with **va_list**, which is defined as the type of **va_alist**).

3. The **va_dcl** macro appears after the function definition and before the opening left brace of the function. This macro is defined as a complete declaration of the **va_alist** parameter, including the terminating semicolon; therefore, no semicolon should follow **va_dcl**.

4. Within the function, the **va_start** macro sets *arg-ptr* to the beginning of the list of optional arguments passed to the function. The **va_start** macro must be used before **va_arg** is used for the first time. The argument *arg-ptr* must have **va_list** type.

5. The **va_arg** macro does the following:
   - Retrieves a value of *type* from the location given by *arg-ptr*
   - Increments *arg-ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts

   The **va_arg** macro can be used any number of times within the function to retrieve the arguments from the list.

6. After all arguments have been retrieved, **va_end** resets the pointer to **NULL**.

Return Value

The **va_arg** macro returns the current argument; **va_start** and **va_end** do not return values.

See Also

vfprintf, **vprintf**, **vsprintf**

Examples

```
#include <stdio.h>
#include <stdarg.h>

main()
{
 int n;
 n = average(2, 3, 4, -1);        /* Call with 4 arguments */
 printf("Average is: %d\n",n);   /* -1 terminates the list */
 n = average(5, 7, 9, 11, -1);    /* Call with 5 arguments */
 printf("Average is: %d\n",n);   /* -1 terminates the list */
}
```

```
average(first,...)
int first;
{
 int i = 0, count = 0, sum = 0;
 va_list arg_marker;
 va_start(arg_marker, first);
 if (first != -1)
  sum = first;
 else
  return(0);
 count++;
 for (; (i = va_arg(arg_marker,int)) >= 0; sum+=i, count++)
  return (sum/count);
}
```

The example above demonstrates how to pass a variable number of arguments using the ANSI C version.

```
#include <stdio.h>
#include <varargs.h>

main()
{
 int n;
 n = average(2, 3, 4, -1);       /* Call with 4 arguments  */
 printf("Average is: %d\n",n);  /* -1 terminates the list */
 n = average(5, 7, 9, 11, -1);   /* Call with 5 arguments */
 printf("Average is: %d\n",n);  /* -1 terminates the list */
}

average(va_alist)
va_dcl
{
 int i = 0, count = 0, sum = 0;
 va_list arg_marker;
 va_start(arg_marker);
 for (; (i = va_arg(arg_marker,int)) >= 0; sum+=i, count++)
  return(count ?(sum/count) : count);
}
```

The second example, above, shows the first example rewritten for compatibility with the UNIX System V version.

vfprintf – vsprintf

Summary

# include <stdio.h>	
# include <varargs.h>	Required for compatibility with UNIX System V
# include <stdarg.h>	Required for compatibility with proposed ANSI C standard

int vfprintf(*stream*, *format*, *argptr***);**

int vprintf(*format*, *argptr***);**

int vsprintf(*buffer*, *format*, *argptr***);**

FILE ****stream*;**	Pointer to **FILE** structure
char ****buffer*;**	Storage location for output
const char ****format*;**	Format control
va_list *argptr***;**	Pointer to list of arguments

Description

The **vfprintf**, **vprintf**, and **vsprintf** functions format and output data to *stream*, the standard output, or *buffer*, respectively. These functions are similar to their counterparts **fprintf**, **printf**, and **sprintf**, but each accepts a pointer to a list of arguments instead of an argument list.

The *format* has the same form and function as the *format* argument for the **printf** function; see the **printf** reference page for a description of *format.*

The *argptr* parameter has type **va_list**, which is defined in **varargs.h** and **stdarg.h**. The *argptr* parameter points to a list of arguments that are converted and output according to the corresponding format specifications in the format.

Return Value

The **vprintf** and **vsprintf** return value is the number of characters written, not counting the terminating null character. If successful, the **vfprintf** return value is the number of characters written. If an output error occurs, it is a negative value.

See Also

fprintf, **printf**, **sprintf**, **va_ arg**, **va_ end**, **va_ start**

Examples

```
#include <stdio.h>
#include <varargs.h>

main()
 {
  int line = 1;
  char *filename = "EXAMPLE";

  /* Call "error" with a format */
  /* string and two parameters  */
  error("Error: line %d, file  filename);

  /* Call "error" with just a */
  /* format string.           */
  error("Syntax error\n");
 }

error(va_alist)
va_dcl
 {
  char *fmt;
  va_list arg_ptr;

  /* "arg_ptr" points to format string */
  va_start(arg_ptr);

  /* "arg_ptr" points to first argument */
  fmt = va_arg(arg_ptr, char *);
  vprintf(fmt, arg_ptr);
  va_end(arg_ptr);
 }
```

Output:

```
Error: line 1, file EXAMPLE
Syntax error
```

The first example, above, conforms to the UNIX System V standard. It uses **vprintf** to set up an error routine that takes a variable number of arguments and displays the appropriate error messages.

```
#include <stdio.h>
#include <stdarg.h>

main()
 {
  int line = 1;
  char *filename = "EXAMPLE";
  /* Call "error" with a format */
  /*  string and two parameters */
  error("Error: line %d, file %s\n", line, filename);

  /* Call "error" with just */
  /*  a format string.      */
  error("Syntax error\n");
 }

error(fmt)
char *fmt;
 {
  va_list arg_ptr;
  va_start(arg_ptr, fmt);     /* "arg_ptr" points to */
                              /*format string */
  vprintf(fmt, arg_ptr);
  va_end(arg_ptr);
 }
```

Output:

```
Error: line 1, file EXAMPLE
Syntax error
```

The second example, above, shows the first example rewritten to conform to the ANSI C standard.

_wrapon

Summary

#include <graph.h>

short far _wrapon(*option*);
short *option*; Wrap condition

Description

The **_wrapon** function controls whether text wraps to a new line or is simply clipped when the text output reaches the edge of the defined text window. The *option* argument can be one of the following manifest constants:

Constant	Meaning
_GWRAPOFF	Truncates lines at window border
_GWRAPON	Wraps lines at window border

Return Value

The function returns the previous value of *switch*. There is no error return.

See Also

_settextwindow

Example

```
#include <stdio.h>
#include <graph.h>

char buffer[ 255 ];
```

```
main()
{
 struct rccoord rcoord;
 int oldcolor;
 /* Set text window to upper half of screen */
 _settextwindow(1, 1, 14, 80 );
 _wrapon(_GWRAPOFF);      /* Turn wrapping off */
 oldcolor = _gettextcolor(); /* Save original color */
 _settextcolor( oldcolor - 1 );
 _settextposition( 1, 1 );
 _outtext("Upper Left corner");
 rcoord = _gettextposition();
 rcoord.row++;
 sprintf(buffer, "Row=%d, Col=%d", rcoord.row, rcoord.col);
 _settextposition( rcoord.row, rcoord.col );
 _outtext( buffer );
 _settextposition( 15, 40);
 _settextcolor( oldcolor );    /* Recover original color */
 _outtext("This should be on the last line, it is out of the window");
 while (!kbhit()); /* wait for key before resetting screen */
 _setvideomode (_DEFAULTMODE);
}
```

This program calls **_wrapon** to disable line wrapping.

write

Summary

#include <io.h>	Required only for function declarations
int write(*handle***,** *buffer***,** *count***);**	
int *handle***;**	Handle referring to open file
char * *buffer***;**	Data to be written
unsigned int *count***;**	Number of bytes

Description

The **write** function writes *count* bytes from *buffer* into the file associated with *handle*. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer (if any) is increased by the number of bytes actually written.

Return Value

The **write** function returns the number of bytes actually written. The return value may be positive but less than *count* (for example, when **write** runs out of disk space before *count* bytes are written).

A return value of –1 indicates an error. In this case, **errno** is set to one of the following values:

Value	Meaning
EBADF	Invalid file handle or file not opened for writing
ENOSPC	No space left on device

If you are writing more than 32K (the maximum size for type **int**) to a file, the return value should be of type **unsigned int**. (See the example that follows.) However, the maximum number of bytes that can be written to a file at one time is 65,534, since 65,535 (or 0xFFFF) is indistinguishable from –1, and so would return an error.

If the file is opened in text mode, each line-feed character is replaced with a carriage-return–line-feed pair in the output. The replacement does not affect the return value.

See Also

fwrite, **open**, **read**

Note

> When writing writes to files opened in text mode, the **write** function treats a CTRL+Z character as the logical end-of-file. When writing to a device, **write** treats a CTRL+Z character in the buffer as an output terminator.

Example

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

char buffer[6000] = "This is a test of 'write' function";

main()
 {
  int fh;
  unsigned int nbytes = 60000, byteswritten;

  if ((fh = open("c:/data/conf.dat",O_WRONLY)) == -1)
   {
    perror("Open failed on output file");
    exit(1);
   }

  if ((byteswritten = write(fh, buffer, nbytes)) == -1)
    perror("");
  else
    printf("Wrote %u bytes to file\n", byteswritten);
 }
```

This program opens a file for output and uses **write** to write 60,000 bytes to the file.

APPENDIXES

APPENDIX A
ERROR MESSAGES

A.1 Introduction

This appendix lists and describes the values to which the **errno** variable can be set when an error occurs in a call to a library routine. Note that only some routines set the **errno** variable. The reference pages for the routines that set **errno** upon error explicitly mention the **errno** variable. (The reference pages are located in Part 2 of this manual.)

An error message is associated with each **errno** value. This message, along with a user-supplied message, can be printed by using the **perror** function.

The value of **errno** reflects the error value for the last call that set **errno**. The **errno** value is not automatically cleared by later successful calls. Thus, to obtain accurate results, you should test for errors and print error messages, if desired, immediately after a call.

The include file **errno.h** contains the definitions of the **errno** values. However, not all of the definitions given in **errno.h** are used under MS-DOS. The full set of values is provided in the include file to maintain compatibility with the XENIX and UNIX include files having the same name.

This appendix lists only the **errno** values used under MS-DOS. For the complete listing of **errno** values, see the **errno.h** include file.

Also listed in this appendix are the errors produced by math routines when an error occurs. These errors correspond to the exception types defined in **math.h** and returned by the **matherr** function when a math error occurs.

A.2 errno Values

Table A.1 gives the **errno** values used on MS-DOS, the system error message corresponding to each value, and a brief description of the circumstances that cause the error.

Table A.1
errno Values and Their Meanings

Value	Message	Description
E2BIG	Arg list too long	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EACCES	Permission denied	Access denied: the file's permission setting does not allow the specified access. This error can occur in a variety of circumstances; it signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes. For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under MS-DOS Versions 3.0 and later, **EACCES** may also indicate a locking or sharing violation. The error can also occur in an attempt to rename a file or directory or to remove an existing directory.
EBADF	Bad file number	The specified file handle is not a valid file-handle value or does not refer to an open file; or an attempt was made to write to a file or device opened for read-only access (or vice versa).
EDEADLOCK	Resource deadlock would occur	Locking violation: the file cannot be locked after 10 attempts (MS-DOS Versions 3.0 and later only).
EDOM	Math argument	The argument to a math function is not in the domain of the function.
EEXIST	File exists	The **O_CREAT** and **O_EXCL** flags are specified when opening a file, but the named file already exists.

Table A.1 *(continued)*

Value	Message	Description
EINVAL	Invalid argument	An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file.
EMFILE	Too many open files	No more file handles are available, so no more files can be opened.
ENOENT	No such file or directory	The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path name does not specify an existing directory.
ENOEXEC	Exec format error	An attempt is made to execute a file that is not executable or that has an invalid executable file-format.
ENOMEM	Not enough core	Not enough memory is available. This message can occur when insufficient memory is available to execute a child process or when the allocation request in an **sbrk** or **getcwd** call cannot be satisfied.
ENOSPC	No space left on device	No more space for writing is available on the device (for example, the disk is full).
ERANGE	Result too large	An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the path-name argument to the **getcwd** function is longer than expected).
EXDEV	Cross-device link	An attempt was made to move a file to a different device (using the **rename** function).

A.3 Math Errors

The following errors can be generated by the math routines of the C run-time library. These errors correspond to the exception types defined in **math.h** and returned by the **matherr** function when a math error occurs; see the **matherr** reference page in Part 2 of this manual for details.

Error	Description
DOMAIN	An argument to the function is outside the domain of the function.
OVERFLOW	The result is too large to be represented in the function's return type.
PLOSS	A partial loss of significance occurred.
SING	Argument singularity: an argument to the function has an illegal value (for example, the value 0 is passed to a function that requires a nonzero value).
TLOSS	A total loss of significance occurred.
UNDERFLOW	The result is too small to be represented. (This condition is not currently supported.)

APPENDIX B
COMMON LIBRARIES

B.1 Introduction

This appendix lists and describes routines from the Microsoft C Run-Time Library for MS-DOS that operate compatibly with C library routines on XENIX systems. The routines provide an identical interface to a set of operations useful on both XENIX and MS-DOS.

The XENIX and MS-DOS common library routines operate compatibly with UNIX library routines as well. In addition, the Microsoft C Run-Time Library for MS-DOS contains several routines that are compatible with UNIX System V routines but that are not currently implemented on XENIX.

With the exception of error returns, the math functions in the Microsoft C Run-Time Library for MS-DOS operate compatibly with the XENIX routines of the same names. Error returns for most math routines in the MS-DOS library have been upgraded for compatibility with UNIX System V math-error handling.

B.2 Run-Time Routines

The sections below list routines from the MS-DOS C library that are compatible with XENIX and UNIX System V routines. Routines specific to the MS-DOS environment are also listed.

B.2.1 Routines Common to MS-DOS and XENIX

The following is a list of the routines common to MS-DOS and XENIX:

abort[1]
abs
access[1]
acos[2]
asctime
asin[2]
assert
atan[2]
atan2[2]
atof
atoi
atol
bessel[3]
bsearch
cabs
calloc
ceil
chdir[1]
chmod[2]
chsize
clearerr
close
cos[2]
cosh[2]
creat[1]
ctime
difftime
dup
dup2
ecvt
execl[1]
execle[1]
execlp[1]
execv[1]
execve[1]
execvp[1]
execvpe[1]
exit
exp
fabs
fclose
fcvt
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
floor
fmod
fopen[1]
fprintf
fputc
fputs
fread[1]
free
freopen[1]
frexp
fscanf
fseek[1]
fstat[1]
ftell[1]
ftime[1]
fwrite[1]
gcvt
getchar
getcwd
getenv
getpid[1]
gets
getw
gmtime
hypot
isalnum
isalpha
isascii
iscntrl
isdigit
isgraph

islower
isprint
ispunct
isspace
isupper
isxdigit
ldexp[2]
lfind
localtime
locking[1]
log[4]
log10[4]
longjmp
lsearch
lseek[1]
malloc
mktemp
modf
onexit
perror
pow[2]
printf
putc
putchar
putenv
puts
putw
qsort
rand
read[1]
realloc
rewind
rmtmp
sbrk
scanf
setbuf
setjmp
setvbuf
signal[1]
sin[2]
sinh[2]
sprintf
sqrt[2]
srand
sscanf
stat[1]
strcat
strchr
strcmp
strcpy
strcspn
strdup
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strtod
strtok
strtol
swab
system[1]
tan[2]
tanh[2]
tempnam
time
tmpfile
tmpnam
toascii
tolower
_tolower
toupper
_toupper
tzset
umask[1]
ungetc
unlink[2]
utime[1]
vfprintf
vprintf
vsprintf
write[1]

B.2.2 Routines Common to MS-DOS and UNIX System V

The XENIX-compatible routines listed in the previous section are also compatible with the routines of the same names in UNIX System V environments. In addition, the following MS-DOS routines are compatible with UNIX System V routines by the same name. These routines are not implemented on XENIX.

alloca
matherr
memccpy
memchr
memcmp
memcpy
memicmp
memset
putenv

Note that most of the math functions in the MS-DOS library implement error handling in the same manner as the UNIX System V routines of the same name. The math routines in the list of common routines for MS-DOS and XENIX (see Section B.2.1) that implement System V-style error handling are footnoted.

---

[1] Operates differently or has different meaning under MS-DOS than under XENIX. The differences are detailed in Section B.5.

[2] Implements UNIX System V-style error returns.

[3] The **bessel** routine does not correspond to a single function, but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. They all implement Unix System V-style error returns.

[4] Implements ANSI-compatible **errno** return values.

B.2.3 Routines Specific to MS-DOS

The routines listed below are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these routines.

_arc
bdos
_bios_disk
_bios_equiplist
_bios_keybrd
_bios_memsize
_bios_printer
_bios_serialcom
_bios_timeofday
cgets
_chain_intr
_clear87
_clearscreen
_control87
cprintf
cscanf
dieeetomsbin
_disable
_displaycursor
dmsbintoieee
_dos_allocmem
_dos_close
_dos_creat
_dos_creatnew
_dos_findfirst
_dos_findnext
_dos_freemem
_dos_getdate
_dos_getdiskfree
_dos_getdrive
_dos_getfileattr
_dos_getftime
_dos_gettime
_dos_getvect
_dos_keep
_dos_open
_dos_read
_dos_setblock
_dos_setdate
_dos_setdrive
_dos_setfileattr
_dos_setftime
_dos_settime
_dos_setvect
_dos_write
_dosexterr
_ellipse
_enable
eof
_exit
fcloseall
_ffree
fgetchar
_fheapchk
_fheapset
_fheapwalk
fieeetomsbin
filelength
_floodfill
flushall
_fmalloc
fmsbintoieee
_fmsize
FP_OFF
FP_SEG
_fpreset
fputchar
_freect
_getbkcolor
getch
getche
_getcolor
_getcursorposition
_getfillmask
_getimage
_getlinestyle
_getlogcoord
_getphyscoord
_getpixel
_gettextcolor
_gettextposition
_getvideoconfig
halloc
_harderr
_hardresume
_hardretn
_heapchk
_heapset
_heapwalk
hfree
_imagesize
inp
inpw
int86
int86x
intdos
intdosx
isatty
itoa
kbhit
labs
_lineto

_lrotl
_lrotr
ltoa
_makepath
max
_memavl
min
mkdir
movedata
_moveto
_msize
_nfree
_nheapchk
_nheapset
_nheapwalk
_nmalloc
_nmsize
outp
outpw
_outtext
_pie
putch
_putimage
_rectangle
_remapallpalette
_remappalette
remove
rename
rmdir
_rotl
_rotr
_searchenv
segread
_selectpalette
_setactivepage
_setbkcolor
_setcliprgn
_setcolor
_setfillmask
_setlinestyle
_setlogorg
setmode
_setpixel
_settextcolor
_settextposition
_settextwindow
_setvideomode
_setviewport
_setvisualpage
sopen
spawnl
spawnle
spawnlp
spawnlpe
spawnv
spawnve
spawnvp
spawnvpe
_splitpath
stackavail
_status
strcmpi
_strdate
strlwr
strncmpi
strnicmp
strnset
strrev
strset
strstr
_strtime
strupr
tell
ultoa
ungetch
_wrapon

B.2.4 ANSI Library

The Microsoft C Run-Time Library includes routines that conform to the Draft Proposed ANSI Standard (ANSI). These routines are listed below. Programs which must strictly adhere to ANSI should use only these routines.

abort
abs
acos
asctime
asin
assert
atan
atan2
atexit
atof
atoi
atol
bsearch
calloc
ceil
clearerr
clock
cos
cosh
ctime
difftime
div
exit
exp
fabs
fclose
feof
ferror
fflush
fgetc
fgetpos
fgets
floor
fmod
fopen
fprintf
fputc
fputs
fread
free
freopen
frexp
fscanf
fseek
fsetpos
ftell
fwrite
getc
getchar
getenv
gets
gmtime
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
labs
ldexp
ldiv
localtime
log
log10
longjmp
malloc
memchr
memcmp
memcpy
memmove

memset
mktime
modf
perror
pow
printf
putc
putchar
puts
qsort
raise
rand
realloc
remove
rename
rewind
scanf
setbuf
setjmp
setvbuf
signal
sin
sinh
sprintf
sqrt
srand
sscanf
strcat
strchr
strcmp
strcpy
strcspn
strerror
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
_strtime
strtod
strtok
strtol
strtoul
system
tan
tanh
tempnam
tmpfile
tolower
toupper
ungetc
va_arg
va_end
va_start
vfprintf
vprintf
vsprintf

B.3 Global Variables

The sections below list global variables used in the MS-DOS C library that are also used in XENIX and UNIX environments. The variables specific to the MS-DOS environment are also listed.

B.3.1 Variables Common to MS-DOS and XENIX

The following is a list of global variables used in the run-time library and available in both the MS-DOS and XENIX environments:

daylight
environ
errno
sys_errlist
sys_nerr
timezone
tzname

Note

> Not all values of **errno** available on XENIX are used by the MS-DOS run-time library.

B.3.2 Variables Common to MS-DOS and UNIX System V

The XENIX-compatible global variables listed in the Section B.3.1 are also available in UNIX System V environments. There are no additional variables common to MS-DOS and UNIX System V.

B.3.3 Variables Specific to MS-DOS

The following global variables are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these variables.

_doserrno
_fmode
_osmajor
_osminor
_psp

B.4 Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. Include files provided with the MS-DOS C library are compatible with include files of the same name on XENIX and UNIX systems. Some additional include files are compatible with include files of the same name in UNIX System V environments.

Sections B.4.1 and B.4.2 list the MS-DOS include files that are compatible with XENIX and UNIX System V. The include files that apply only to MS-DOS environments are listed in Section B.4.3.

B.4.1 Include Files Common to MS-DOS and XENIX

The following MS-DOS include files are compatible with the XENIX (and UNIX) include files of the same name:

asserth
ctypeh
errnoh
fcntlh
mathh
setjmph
signalh
stdioh
sys\lockingh
sys\stath
sys\timebh
sys\typesh
timeh

B.4.2 Include Files Common to MS-DOS and UNIX System V

The XENIX-compatible include files listed in Section B.4.1 are also compatible with the include files of the same names in UNIX System V environments. In addition, the names of the following MS-DOS include files correspond to UNIX System V include files; however, the MS-DOS include files may not contain all the constants and types defined in the corresponding UNIX System V include files.

malloc.h
memory.h
search.h
string.h
varargs.h

B.4.3 Include Files Specific to MS-DOS

The following include files are used only in MS-DOS environments and do not have counterparts on XENIX and UNIX systems:

conio.h
direct.h
dos.h
graph.h
io.h
process.h
share.h
stdarg.h
stdlib.h
sys\utime.h

B.4.4 ANSI Include Files

The include files necessary to use the ANSI run-time library are listed below:

assert.h
ctype.h
float.h
limits.h
math.h
setjmp.h
signal.h
stdarg.h
stdio.h
stdlib.h
string.h
time.h

B.5 Differences Between Routines Common to MS-DOS and XENIX

Sections B.5.1 – B.5.26 explain how the MS-DOS routines in the common library for XENIX and MS-DOS differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of MS-DOS functions provided in the reference section (Part 2 of this manual) and with the descriptions of the XENIX routines in the appropriate XENIX manual.

B.5.1 abort

The MS-DOS version of the **abort** routine terminates the process by a call to **raise(SIG_ABRT)**. Control is returned to the parent (calling) process with an exit status of 3 and the following message is printed to standard error:

```
Abnormal program termination
```

No core dump occurs on MS-DOS.

B.5.2 access

The **access** routine checks the access to a given file. Under MS-DOS, the real and effective user IDs are nonexistent. The permission (access) setting can be any combination of the following values:

Value	Meaning
04	Read
02	Write
00	Check for existence

The "Execute" access mode (01) is not implemented.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

B.5.3 chdir

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.4 chmod

The **chmod** routine can set the "owner" access permissions for a given file, but all other permission settings are ignored. The mode argument can be any one of the constant expressions shown in the left-most column below; the equivalent XENIX value is shown in the right-most column:

Constant Expression	Meaning	XENIX Value
S_IREAD	Read by owner	0400
S_IWRITE	Write by owner	0200
S_IREAD ¦ S_IWRITE	Read and write by owner	0000

The **S_IREAD** and **S_IWRITE** constants are defined in the **sys\stat.h** include file. Note that the **OR** operator (¦) is used to combine these constants to form read and write permission.

If write permission is not given, the file is treated as a read-only file. Giving write-only permission is allowed, but has no effect; under MS-DOS, all files are readable.

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.5 creat

The **creat** routine creates a new file or prepares an existing file for writing. If the file is created successfully, the access permissions are set as defined by the mode argument. Only "owner" permissions are allowed (see **chmod** above).

In case of error, only the **EACCES**, **EMFILE**, and **ENOENT** values may be returned for **errno** on MS-DOS.

Use of the **open** routine is preferred over **creat** when creating or opening files in both MS-DOS and XENIX environments.

B.5.6 exec

The MS-DOS versions of the **execl**, **execle**, **execlp**, **execlpe**, **execv**, **execve**, **execvpe**, and **execvp** routines overlay the calling process, as in the XENIX environment. If there is not enough memory for the new process, the **exec** routine fails and returns to the calling process. Otherwise, the new process begins execution.

Under MS-DOS, the **exec** routines *do not* perform the following functions:

- Use the close-on-exec flag to determine open files for the new process.
- Disable profiling for the new process (profiling is not available under MS-DOS).
- Pass signal settings to the child process. Under MS-DOS, all signals (including signals set to be ignored) are reset to the default in the child process.

The combined size of all arguments (including the program name) in an **exec** routine under MS-DOS must not exceed 128 bytes.

In case of error, the **E2BIG**, **EACCES**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to determine whether it is executable.

B.5.7 fopen, freopen

The MS-DOS versions of the **fopen** and **freopen** routines open stream files just as they do in the XENIX environment. However, under MS-DOS the following additional values for the *type* string are available:

Value	Meaning
t	Opens the file in text mode. Opening a file in this mode causes translation of carriage-return–line-feed (CR-LF) character combinations into a single line feed (LF) on input. Similarly, on output, line feeds are translated into CR-LF combinations.
b	Opens the file in binary mode. This mode suppresses translation.

See the MS-DOS reference pages (in Part 2 of this manual) for the **fopen** and **freopen** routines to obtain more information on the default mode setting.

The MS-DOS and XENIX versions of these routines also differ in their interpretation of append mode (**a** or **a+**). When append mode is specified in the MS-DOS version of **fopen** or **freopen**, the file pointer is repositioned at the end of the file prior to write operations. Thus all write operations take place at the end of the file.

In the XENIX versions, all write operations take place at the current position of the file pointer. In append mode, the file pointer is initially positioned at the end of the file, but if the file pointer is later repositioned, write operations take place at the new position rather than at the end of the file.

B.5.8 fread

The MS-DOS **fread** routine uses the low-level **read** function to carry out read operations. If the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting the end of the file.

B.5.9 fseek

Both the MS-DOS and XENIX versions of the **fseek** routine move the file pointer to the given position. However, for streams opened in text mode, the MS-DOS version of **fseek** has limited use because CR-LF translations can cause **fseek** to produce unexpected results. Only two **fseek** operations are guaranteed to work on streams opened in text mode: seeking with an offset of 0 relative to any of the origin values, and seeking from the beginning of the file with an offset value returned from a call to **ftell**.

B.5.10 fstat

MS-DOS does not make as much information available for file handles as it does for full path names; thus the MS-DOS version of **fstat** returns less useful information than does the **stat** routine. The MS-DOS **fstat** routine can detect device files, but it must not be used with directories.

The structure returned by **fstat** contains the following members:

Member	**Meaning**
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).
st_dev	Either the drive number of the disk containing the file, or the file handle in the case of a device (same as **st_rdev**).

st_gid	Not used.
st_ino	Not used.
st_mode	User read and write bits reflect the file's permission setting. The **S_IFCHR** bit is set for a device; otherwise, the **S_IFREG** bit is set.
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_nlink	Always 1.
st_rdev	Either the drive number of the disk containing the file, or the file handle in the case of a device (same as **st_dev**).
st_size	Size, in bytes, of the file.
st_uid	Not used.

In case of error, only the **EBADF** value may be returned for **errno** on MS-DOS.

B.5.11 ftell

Both the MS-DOS and XENIX versions of the **ftell** routine get the current file-pointer position. In MS-DOS, however, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, since text mode causes CR-LF translation. The **ftell** routine can be used in conjunction with the **fseek** routine to remember and return to file locations correctly. If you want the actual offset to a file position, open the stream in binary mode and perform type conversions as necessary.

B.5.12 ftime

Unlike the system time on XENIX systems, the MS-DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of an MS-DOS environment variable named **TZ** to determine the time zone. The user can set the default time zone by setting the **TZ** variable. If **TZ** is not explicitly set, the default time zone corresponds to the Pacific time zone. See the reference page for **tzset** in Part 2 of this manual for details on the **TZ** variable.

B.5.13 fwrite

The MS-DOS **fwrite** routine uses the low-level **write** function to carry out write operations. If the file is opened in text mode, every line-feed (LF) character in the output is replaced by a carriage-return–line-feed (CR-LF) pair before being written. This does not affect the return value.

B.5.14 getpid

The **getpid** routine returns a process-unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process identification returned by **getpid** in the XENIX environment.

B.5.15 locking

The MS-DOS and XENIX versions of the **locking** routine differ in several respects, as listed below:

- On MS-DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. Thus, setting **LK_RLCK** in the **locking** call is equivalent to setting **LK_LOCK**, and setting **LK_NBRLCK** is equivalent to setting **LK_NBLCK**.

- On MS-DOS, specifying **LK_LOCK** or **LK_RLCK** will *not* cause a program to wait until the specified region of a file is unlocked. Instead, up to ten attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the **locking** function returns an error value.

  On XENIX, if the first attempt at locking fails, the locking process "sleeps" (suspends execution) and periodically "wakes" to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.

- On MS-DOS, locking of overlapping regions of a file is not allowed.

- On MS-DOS, if more than one region of a file is locked, only one region can be unlocked at a time, and the region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

B.5.16 log, log10

Passing a 0 to **log** or **log10** sets the **errno** variable to **EDOM** on XENIX, instead of setting it to **ERANGE** as it does on MS-DOS.

B.5.17 lseek

In case of error, only the **EBADF** and **EINVAL** values may be returned for **errno** on MS-DOS.

B.5.18 open

Both the MS-DOS and XENIX versions of the **open** routine open a file by its handle. However, with MS-DOS, two additional *oflag* values (**O_BINARY** and **O_TEXT**) are available and the **O_NDELAY** and **O_SYNCW** values are not available.

The **O_BINARY** flag causes the file to be opened in binary mode, regardless of the default mode setting. Similarly, the **O_TEXT** flag causes the file to be opened in text mode.

In case of error, only the **EACCES**, **EEXIST**, **EMFILE**, and **ENOENT** values may be used for **errno** on MS-DOS.

B.5.19 read

Both the MS-DOS and XENIX versions of the **read** routine read characters from the file given by a file handle. However, if the file has been opened in text mode, the MS-DOS version of **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus, the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting an end-of-file condition.

In case of error, only the **EBADF** value may be used for **errno** on MS-DOS.

B.5.20 signal

The MS-DOS version of the **signal** routine can only handle the **SIGINT**, **SIGFPE**, **SIGABRT**, **SIGILL**, and **SIGSEGV** signals. In MS-DOS, **SIGINT** is defined to be INT 23H (the signal), **SIGFPE** corresponds to floating-point exceptions that are not masked, **SIGABRT** is the default **abort** handler, and **SIGILL** and **SIGSEGV** are undefined, but provided for ANSI compatibility.

On MS-DOS, child processes executed through the **exec** or **spawn** routines do not inherit the signal settings of the parent process. All signal settings (including signals set to be ignored) are reset to the default settings in the child process.

The MS-DOS version of **signal** uses only **EINVAL** for **errno**.

B.5.21 stat

The **stat** routine returns a structure defining the current status of the given file or directory. The structure members returned by **stat** have the following names and meanings on MS-DOS:

Value	Meaning
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).
st_dev	Drive number of the disk containing the file (same as **st_rdev**).
st_gid	Not used.
st_ino	Not used.
st_mode	User read and write bits reflect the file's permission setting. The **S_IFDIR** bit is set for a device; otherwise, the **S_IFREG** bit is set.
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_nlink	Always 1.
st_rdev	Drive number of the disk containing the file (same as **st_ dev**).
st_size	Size, in bytes, of the file.
st_uid	Not used.

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.22 system

The **system** routine passes the given string to the operating system for execution. For MS-DOS to execute this string, the full path name of the directory containing it must be assigned to the environment variable. If the string is a **NULL**, the system searches for **COMMAND.COM**.

The **system** call returns an error if the string cannot be found using these variables. Where a null pointer is passed, it sets **errno** to **ENOENT** and returns 0 if it cannot find **COMMAND.COM**, and 1 if it can. In case of error, only the **E2BIG**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on MS-DOS.

B.5.23 umask

The **umask** routine can set a mask for "owner" read and write access permissions only. All other permissions are ignored. (See the discussion of the **access** routine above for details.)

B.5.24 unlink

The MS-DOS version of the **unlink** routine always deletes the given file. Since MS-DOS does not implement multiple "links" to the same file, unlinking a file is the same as deleting it.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

B.5.25 utime

The MS-DOS **utime** routine sets the file modification time only; MS-DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to set the modification time.

B.5.26 write

Both the MS-DOS and XENIX versions of the **write** routine write a specified number of characters to the file named by the given file handle. However, in the MS-DOS version, if the file has been opened in text mode, every line-feed (LF) character in the output is replaced by a carriage-return–line-feed (CR-LF) pair before being written. This does not affect the return value.

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on MS-DOS.

INDEX