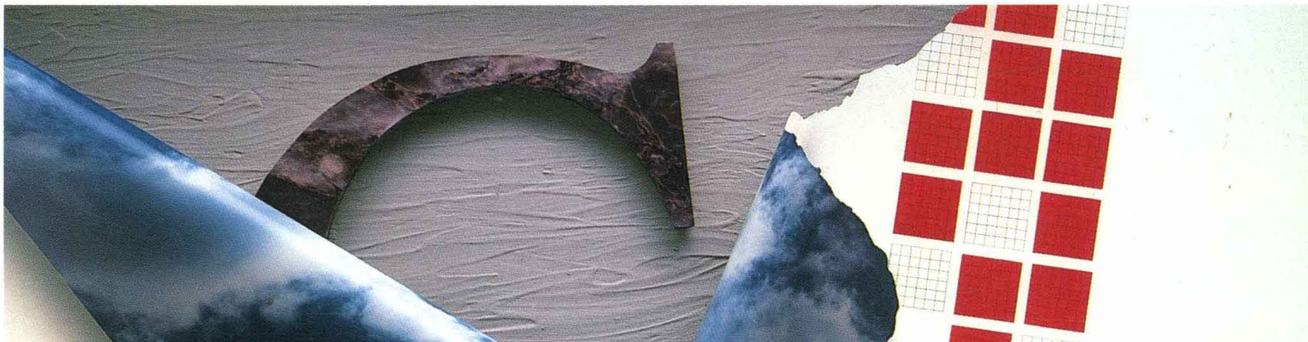
Covers version 6

Microsoft® C
Run-Time Library Reference



Microsoft Languages Library

Microsoft PRESS®

Covers version 6

Microsoft® C
Run-Time Library Reference

Microsoft Languages Library

Microsoft PRESS®

Written, edited, and produced by Microsoft Corporation

Distributed by Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way, Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data

Microsoft C run-time library reference.

Includes index.
1. C (Computer program language) 2. Microsoft C
(Computer program) 3. Macro instructions (Electronic
computers) I. Microsoft.
QA76.73.C15M52 1990 005.13'3 89-12240
ISBN 1-55615-225-6

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 HCHC 3 2 1 0 9

Distributed to the book trade in Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging in Publication Data available.

Writers:	Editors:	Sample Programs:
Phil Nelson	Amanda Clark	Bruce McKinney
Terry Ward	Moira Macdonald	
	Marjorie Manwaring	
	Bill Nolan	

Contents

Introduction v

About the C Run-Time Library v
About This Book vii
Other Books of Interest vii
Document Conventions ix

PART 1 Overview

Chapter 1 Using C Library Routines 5

1.1 Calling Library Routines 5
1.2 Using Header Files 6
1.3 File Names and Path Names 9
1.4 Choosing Between Functions and Macros 10
1.5 Stack Checking on Entry 12
1.6 Handling Errors 13
1.7 Operating-System Considerations 14
1.8 Floating-Point Support 15
1.9 Using Huge Arrays with Library Functions 16

Chapter 2 Run-Time Routines by Category 19

2.1 Buffer Manipulation 20
2.2 Character Classification and Conversion 21
2.3 Data Conversion 22
2.4 Directory Control 23
2.5 File Handling 23
2.6 Graphics 25
2.7 Input and Output 35
2.8 Internationalization 45
2.9 Math 45
2.10 Memory Allocation 48

2.11 Process and Environment Control 51
2.12 Searching and Sorting 55
2.13 String Manipulation 55
2.14 System Calls 56
2.15 Time 60
2.16 Variable-Length Argument Lists 62

Chapter 3 Global Variables and Standard Types 63

3.1 _amblksiz 63
3.2 daylight, timezone, tzname 64
3.3 _doserrno, errno, sys_errlist, sys_nerr 65
3.4 _fmode 66
3.5 _osmajor, _osminor, _osmode, _osversion 67
3.6 environ 67
3.7 _psp 68
3.8 Standard Types 68

PART 2 Run-Time Functions

About the Run-Time Reference 75

Alphabetic Function Reference 76

Index 829

Introduction

The Microsoft® C Run-Time Library is a set of over 500 ready-to-use functions and macros designed for use in C programs. The run-time library makes programming easier by providing

- Fast and efficient routines to perform common programming tasks (such as string manipulation), sparing you the time and effort needed to write such routines
- Reliable methods of performing operating-system functions (such as opening and closing files)

The C run-time library is important because it provides basic functions not provided by the C language itself. These functions include input and output, memory allocation, process control, graphics, and many others.

This book describes the Microsoft C run-time library routines included with the Microsoft Professional Development System version 6.0. These comprise all of the routines included with earlier versions of Microsoft C, as well as many new routines.

NOTE *Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

About the C Run-Time Library

The Microsoft C run-time library contains a number of new routines and features which support American National Standards Institute (ANSI) C compatibility, OS/2 and XENIX® programming, and sophisticated graphics programming.

To ease the task of transporting programs from one operating system to another, the description of each library routine includes compatibility boxes, which show at a glance whether the routine is compatible with ANSI C, MS-DOS, OS/2, UNIX®, and XENIX. (In this book, references to XENIX systems also encompass UNIX and other UNIX-like systems.)

ANSI C Compatibility

The C run-time library routines are designed for compatibility with the ANSI C standard, which Microsoft C compilers support. The major innovation of ANSI C is to permit argument-type lists in function prototypes (declarations). Given the information in the function prototype, the compiler can check later references to the function to make sure that the references use the correct number and type of arguments and the correct return value.

To take advantage of the compiler's type-checking ability, the include files that accompany the C run-time library have been expanded. In addition to the definitions and declarations required by library routines, the include files now contain function declarations with argument-type lists. Several new include files have also been added. The names of these files are chosen to maximize compatibility with the ANSI C standard and with XENIX and UNIX names.

OS/2 and XENIX® Programming

Microsoft C run-time library routines are designed to maintain maximum compatibility between MS-DOS, OS/2, and XENIX or UNIX systems. The library offers a number of operating-system interface routines that allow you to take advantage of specific DOS and OS/2 features.

Most of the functions in the C library for DOS and OS/2 are compatible with like-named routines in the C library for XENIX. For additional compatibility, the math library functions have been extended to provide exception handling in the same manner as the UNIX System V math functions.

Expanded Graphics Library

The Microsoft C run-time library now contains over one hundred graphics routines. The core of this library consists of several dozen low-level graphics routines, which allow your programs to select video modes, set points, draw lines, change colors, and draw shapes such as rectangles and ellipses. You can display real-valued data, such as floating-point values, within windows of different sizes by using various coordinate systems.

Recent additions to the graphics library include presentation graphics and fonts. The presentation-graphics library provides powerful tools for adding presentation-quality graphics to your programs. These routines can display data as a variety of graphs, including pie charts, bar and column charts, line graphs, and scatter diagrams.

The fonts library allows your programs to display various styles and sizes of text in graphics images or charts. You can use font-manipulation routines with any graphics routines that display text, including presentation graphics.

About This Book

This book assumes that you understand the C language and know how to compile and link programs. If you have questions about these subjects, consult your compiler documentation.

This book has two parts. Part 1, "Overview," introduces the Microsoft C library. It describes general rules for using the library and summarizes the main categories of library routines. Part 1 contains the following chapters:

- Chapter 1, "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 1 when you have questions about library procedures.
- Chapter 2, "Run-Time Routines by Category," lists the C library routines by category and discusses considerations that apply to each category. This chapter makes it easy to locate routines by task. Once you find the routine you want, turn to the reference page in Part 2 for a detailed description.
- Chapter 3, "Global Variables and Standard Types," describes variables and types that are used by library routines. Global variables and standard types are also described in the reference descriptions of the routines that use them.

Part 2, "Run-Time Functions," describes the library routines in alphabetical order. Once you are familiar with the C library rules and procedures, you will probably use this part most often.

Other Books of Interest

This book provides a guide to the C run-time library provided with the Microsoft C Professional Development System version 6.0.

The following books cover a variety of topics that you may find useful. They are listed only for your convenience. With the exception of its own publications, Microsoft does not endorse these books or recommend them over others on the same subject.

- Barkakati, Nabajyoti. *The Waite Group's Microsoft C Bible*. Indianapolis, IN: Howard W. Sams, 1988.

  A topical guide to the Microsoft C run-time library. A similar volume is available for the Microsoft QuickC® product.

- Campbell, Joe. *C Programmer's Guide to Serial Communications*. Indianapolis, IN: Howard W. Sams & Company, 1987.

  A comprehensive guide to the specialized area of serial communication programming in C.

- Hansen, Augie. *Proficient C: The Microsoft Guide to Intermediate & Advanced C Programming*. Redmond, WA: Microsoft Press, 1987.

  An intermediate-level guide to C programming.

- Harbison, Samuel P., and Guy L. Steele, Jr. *C: A Reference Manual*, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 1987.

  A comprehensive guide to the C language and the standard library.

- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

  The first edition of this book is the classic definition of the C language. The second edition includes new information on the proposed ANSI C standard.

- Lafore, Robert. *Microsoft C Programming for the IBM*. Indianapolis, IN: Howard W. Sams & Company, 1987.

  The first half of this book teaches C. The second half concentrates on specifics of the PC environment, such as BIOS calls, memory, and video displays.

- Mark Williams Company. *ANSI C: A Lexical Guide*. Englewood Cliffs, NJ: Prentice Hall, 1988.

  A dictionary-style guide to the ANSI C standard.

- Plauger, P. J., and Jim Brodie. *Standard C*. Redmond, WA: Microsoft Press, 1989.

  A quick reference guide to the ANSI C implementation by the secretary and chairman of the ANSI-authorized C Programming Language Standards Committee.

- Plum, Thomas. *Reliable Data Structures in C.* Cardiff, NJ: Plum Hall, 1985.

  An intermediate-level look at data structures using the C language.

- Plum, Thomas, and Jim Brodie. *Efficient C.* Cardiff, NJ: Plum Hall, 1985.

  A guide to techniques for increasing the efficiency of C programs.

- Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* New York: Cambridge University Press, 1988.

  A comprehensive look at numerical techniques using the C language.

- Schustack, Steve. *Variations in C: Programming Techniques for Developing Efficient Professional Applications.* Redmond, WA: Microsoft Press, 1985.

  An intermediate-level guide to developing business applications in C.

- Ward, Robert. *Debugging C.* Indianapolis, IN: Que Corporation, 1986.

  An advanced guide to the theory and practice of debugging C programs.

- Wilton, Richard. *Programmer's Guide to PC and PS/2 Video Systems:Maximum Video Performance from the EGA, VGA, HGC, & MCGA.* Redmond, WA: Microsoft Press, 1987.

  An advanced guide to all the PC and PS/2 video modes.

Document Conventions

This book uses the following document conventions :

Example	Description
STDIO.H	Uppercase letters indicate file names, segment names, registers, and terms used at the operating-system command level.
_far	Boldface letters indicate C keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
expression	Words in italics indicate placeholders for information you must supply, such as a file name. Italics are also occasionally used for emphasis in the text.
[[*option*]]	Items inside double square brackets are optional.

#pragma pack {1|2}

Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets surround the braces.

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

This font is used for examples, user input, program output, and error messages in text.

CL *options* [[*files...*]]

Three dots following an item indicate that more items having the same form may appear.

```
while()
{
   .
   .
   .
}
```

A column of three dots tells you that part of the example program has been intentionally omitted.

CTRL+ENTER

Small capital letters are used for the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.

The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.

"argument"

Quotation marks enclose a new term the first time it is defined in text.

```
"C string"
```

Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " and ' ' rather than “ ” and ‘ ’.

Color Graphics Adapter (CGA)

The first time an acronym is used, it is often spelled out.

Special Offer
Companion Disk for Microsoft C Run-Time Library Reference

Microsoft Press has created a companion disk for *Microsoft C Run-Time Library Reference*. This disk, available in 5.25- and 3.5-inch format, contains nearly 300 example programs from the book. You can use code fragments from the companion disk for commercial or personal purposes without infringing on the copyright of the book.

Domestic Ordering Information

To order, use the special reply card bound in the back of the book. If the card has already been used, please send $19.95 (plus sales tax if applicable: CA residents, 5% plus local option tax, CT 8%, FL 6%, IL 5%, KY 5%, MA 5%, MN 6%, MO 4.425%, NJ 6%, NY 4% plus local option tax, SC 5%, TX 6% plus local option tax, WA State 7.8%) and $2.50 per disk set for domestic postage and handling charges. Mail your order to: Microsoft Press, Attn: Companion Disk Offer, 21919 20th Ave SE, Box 3011, Bothell, WA 98041-3011. Please specify 5.25-inch format or 3.5-inch format. Payment must be in U.S. funds. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, or MasterCard. (If paying by credit card, please include both your card number and the expiration date.) Allow 2–3 weeks for delivery.

Foreign Ordering Information (within the U.K., see below)

Follow ordering procedures for domestic ordering and add $6.00 for foreign postage and handling.

U.K. Ordering Information

Send your order in writing along with £18.95 (includes VAT) to: Microsoft Press, 27 Wrights Lane, London W8 5TZ. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA, MasterCard, or Diners Club. (If paying by credit card, please include both your card number and the expiration date.) Specify 5.25-inch format or 3.5-inch format.

Microsoft Press Companion Disk Guarantee

If the disk proves defective, send the defective disk along with your packing slip (or copy) to: Microsoft Press, Consumer Sales, One Microsoft Way, Redmond, WA 98052-6399.

If you have questions or comments about the files on the disk, send them to: Languages User Education, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399.

The Companion Disk for *Microsoft C Run-Time Library Reference* is available only from Microsoft Press.

PART 1

Overview

CHAPTERS

1 Using C Library Routines 5

2 Run-Time Routines by Category 19

3 Global Variables and Standard Types 63

Overview

The first part of this book provides an overview of the run-time library provided with the Microsoft C Professional Development System.

Chapter 1 is a general guide to the use of the run-time library routines.

Chapter 2 lists the routines by category.

Chapter 3 tells how to access global variables and types defined in the run-time library.

CHAPTER 1

Using C Library Routines

This chapter provides basic information about how to use Microsoft C library routines. It also describes some special rules, such as file- and path-name conventions, that apply to particular routines. You should read this chapter before you begin to use C library routines, and you may also want to refer back to it if you have questions about library procedures.

1.1 Calling Library Routines

To use a C library routine, simply call it in your program, just as if it is defined there. For instance, suppose you write the following program and name it SAMPLE.C:

```
#include <stdio.h>
main()
{
   printf( "Microsoft C" );
}
```

The program prints `Microsoft C` by calling the **printf** routine, which is part of the standard C library. Calling a library routine normally involves two groups of files:

1. Header ("include") files that contain declarations and type definitions required by library routines
2. Library files that contain the library routines in compiled form

Header files and library files are both included with Microsoft C. Header files are used when compiling, and library files are used when linking.

You include the necessary header files in your program source code with **#include** directives. The description of each library routine in Part 2, "Reference," tells you what header file the routine requires. Since **printf** requires the STDIO.H header file, the SAMPLE.C program contains the following line:

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

This line causes the compiler to insert the contents of STDIO.H into the source file SAMPLE.C.

After you compile the source file, you link the resulting object (.OBJ) file with the appropriate library (.LIB) file to create an executable (.EXE) file. Your object file contains the name of every routine that your program calls, including library routines. If a routine is not defined in your program, the linker searches for its code in a library file and includes that code in the executable file.

Normally, the code for standard library routines is contained in the "default library" that you create when installing Microsoft C. Since the linker automatically searches the default library, you do not need to specify that library's name when linking your program. The following command links the example program with the default library:

```
link sample,,,;
```

If you call a library routine that is not contained in the default library, you must give the linker the name of the library file that contains the routine. For instance, suppose your program uses a Microsoft C graphics routine and you did not make GRAPHICS.LIB part of your default library when installing Microsoft C. You would then link the program using a line like the following:

```
link sample,,, graphics.lib;
```

For more information about libraries and linking, consult the installation documentation for your compiler.

1.2 Using Header Files

As stated in the previous section, you should include C header files when using library routines. This section describes particular reasons why header files are required.

1.2.1 Including Necessary Definitions

Many C library routines use constants, type definitions, or macros defined in a header file. To use the routine, you must include the header file containing the needed definition(s). The following list gives examples:

Definition	Example
Macro	If a library routine is implemented as a macro, the macro definition appears in a header file. For instance, the **toupper** macro is defined in the header file CTYPE.H.
Manifest constant	Many library routines refer to constants that are defined in header files. For instance, the **open** routine uses constants such as **O_CREAT**, which is defined in the header file FCNTL.H.
Type definition	Some library routines return a structure or take a structure as an argument. For example, stream input/output routines use a structure of type **FILE**, which is defined in STDIO.H.

1.2.2 Including Function Declarations

The Microsoft C header files also contain function declarations for every function in the C library. These declarations are in the style recommended by the ANSI C standard. Given these declarations, the compiler can perform "type checking" on every reference to a library function, making sure that you have used the correct return type and arguments. Function declarations are sometimes called "prototypes," since the declaration serves as a prototype or template for every subsequent reference to the function.

A function declaration lists the name of the function, its return type, and the number and type of its arguments. For instance, below is the declaration of the **pow** library function from the header file MATH.H:

double pow(double *x*, **double** *y* **);**

The example declares that **pow** returns a value of type **double** and takes two arguments of type **double**. Given this declaration, the compiler can check every reference to **pow** in your program to ensure that the reference passes two **double** arguments to **pow** and takes a return value of type **double**.

The compiler can perform type checking only for function references that appear after the function declaration. Because of this, function declarations normally appear near the beginning of the source file, prior to any use of the functions they declare.

Function declarations are especially important for functions that return a value of some type other than **int**, which is the default. For example, the **pow** function returns a **double** value. If you do not declare such a function, the compiler treats its return value as **int**, which can cause unexpected results.

It is also a good practice to provide declarations for functions that you write. If you do not want to type the declarations by hand, you can generate them automatically by using the /Zg compiler option. This option causes the compiler to generate ANSI-standard function declarations for every function defined in the current source file. Redirect this output to a file, then insert the file near the beginning of your source file.

Your program can contain more than one declaration of the same function, as long as the declarations do not conflict. This is important if you have old programs whose function declarations do not contain argument-type lists. For instance, if your program contains the declaration

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

you can later include the following declaration:

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

Because the two declarations are compatible, even though they are not identical, no conflict occurs. The second declaration simply gives more information about function arguments than the second. A conflict would arise, however, if the declarations gave a different number of arguments or gave arguments of different types.

Some library functions can take a variable number of arguments. For instance, the **printf** function can take one argument or several. The compiler can perform only limited type checking on such functions, a factor that affects the following library functions:

- In calls to **cprintf**, **cscanf**, **printf**, and **scanf**, only the first argument (the format string) is type checked.
- In calls to **fprintf**, **fscanf**, **sprintf**, and **sscanf**, only the first two arguments (the file or buffer and the format string) are type checked.
- In calls to **open**, only the first two arguments (the path name and the **open** flag) are type checked.
- In calls to **sopen**, only the first three arguments (the path name, the **open** flag, and the sharing mode) are type checked.

- In calls to **execl**, **execle**, **execlp**, and **execlpe**, only the first two arguments (the path name and the first argument pointer) are type checked.
- In calls to **spawnl**, **spawnle**, **spawnlp**, and **spawnlpe**, only the first three arguments (the mode flag, the path name, and the first argument pointer) are type checked.

1.3 File Names and Path Names

Many library routines take strings representing paths and file names as arguments. If you plan to transport your programs to the XENIX operating system, you should remember that XENIX uses file- and path-name conventions that are different from those used by DOS and OS/2. If you do not plan to transport your programs to XENIX, you can skip this section.

Case Sensitivity

The DOS and OS/2 operating systems are not case sensitive (they do not distinguish between uppercase and lowercase letters). Thus, SAMPLE.C and Sample.C refer to the same file in DOS and OS/2. However, the XENIX operating system is case sensitive. In XENIX, SAMPLE.C and Sample.C refer to different files. To transport programs to XENIX, choose file and path names that work correctly in XENIX, since either case works in DOS and OS/2. For instance, the following directives are identical in DOS and OS/2, but only the second works in XENIX:

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

Subdirectory Conventions

Under XENIX, certain header files are normally placed in a subdirectory named SYS. Microsoft C follows this convention to ease the process of transporting programs to XENIX. If you do not plan to transport your programs, you can place the SYS header files elsewhere.

Path-Name Delimiters

XENIX uses the slash (/) in path names, while DOS and OS/2 use the backslash (\). To transport programs to XENIX, it is advantageous to use path-name delimiters that are compatible with XENIX whenever possible.

1.4 Choosing Between Functions and Macros

This book uses the words "routine" and "function" interchangeably. However, the term "routine" actually encompasses both functions and macros. Because functions and macros have different properties, you should pay attention to which form you are using. The descriptions in the reference section indicate whether routines are implemented as functions or as macros.

Most routines in the Microsoft C library are functions. They consist of compiled C code or assembled Microsoft Macro Assembler (MASM) code. However, a few library routines are implemented as macros that behave like functions. You can pass arguments to library macros and invoke them in the same way you invoke functions.

The main benefit of using macros is faster execution time. A macro is expanded (replaced by its definition) during preprocessing, creating in-line code. Thus, macros do not have the overhead associated with function calls. On the other hand, each use of a macro inserts the same code in your program, whereas a function definition occurs only once regardless of how many times it is called. Functions and macros thus offer a trade-off between speed and size.

Apart from speed and size issues, macros and functions have some other important differences:

- Some macros treat arguments with side effects incorrectly when the macro evaluates its arguments more than once (see the example that follows this list). Not every macro has this effect. To determine if a macro handles side effects as desired, examine its definition in the appropriate header file.
- A function name evaluates to an address, but a macro name does not. Thus, you cannot use a macro name in contexts requiring a function pointer. For instance, you can declare a pointer to a function, but you cannot declare a pointer to a macro.
- You can declare functions, but you cannot declare macros. Thus, the compiler cannot perform type checking of macro arguments as it does of function arguments. However, the compiler can detect when you pass the wrong number of arguments to a macro.
- You must always include the appropriate header file when using a library macro. Every library macro is defined with a **#define** directive in a header file. If you do not include the header file, the macro is undefined.

The following example demonstrates how some macros can produce unwanted side effects. It uses the **toupper** routine from the standard C library.

```
#include <ctype.h>

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

The example increments `a` when passing it as an argument to the **toupper** routine, which is implemented as a macro. It is defined in CTYPE.H:

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

The definition uses the conditional operator (**? :**). The conditional expression evaluates the argument `c` twice: once to check if it is lowercase and again to create the result. This macro evaluates the argument `a++` twice, increasing `a` by 2 instead of 1. As a result, the value operated on by **islower** differs from the value operated on by **_toupper.**

Like some other library routines, **toupper** is provided in both macro and function versions. The header file CTYPE.H not only declares the **toupper** function but also defines the **toupper** macro.

Choosing between the macro version and function version of such routines is easy. If you wish to use the macro version, you can simply include the header file that contains the macro definition. Because the macro definition of the routine always appears after the function declaration, the macro definition normally takes precedence. Thus, if your program includes CTYPE.H and then calls **toupper,** the compiler uses the **toupper** macro:

```
#include <ctype.h>

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

You can force the compiler to use the function version of a routine by enclosing the routine's name in parentheses:

```
#include <ctype.h>

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

Because the name **toupper** is not immediately followed by a left parenthesis, the compiler cannot interpret it as a macro name. It must use the **toupper** function.

A second way to do this is to "undefine" the macro definition with the **#undef** directive:

```
#include <ctype.h>
#undef toupper
```

Since the macro definition no longer exists, subsequent references to **toupper** use the function version.

A third way to make sure the compiler uses the function version is to declare the function explicitly:

```
#include <ctype.h>
int toupper(int _c);
```

Since this function declaration appears after the macro definition in CTYPE.H, it causes the compiler to use the **toupper** function.

1.5 Stack Checking on Entry

For certain library routines, the compiler performs stack checking on entry. (The "stack" is a memory area used for temporary storage.) Upon entry to such a routine, the stack is checked to determine if it has enough room for the local variables used by that routine. If it does, 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 there is not, you might overwrite memory locations in the data segment and receive no warning.

Typically, stack checking is enabled only for functions with large local-variable requirements (more than about 150 bytes), 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 slows execution slightly.

Stack checking is enabled for the following library functions:

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

1.6 Handling Errors

Many library routines return a value that indicates an error condition. To avoid unexpected results, your code should always check such error values and handle all of the possible error conditions. The description of each library routine in the reference section lists the routine's return value(s).

Some library functions do not have a set error return. These include functions that return nothing and functions whose range of return values makes it impossible to return a unique error value. To aid in error handling, some functions in this category set the value of a global variable named **errno**.

If the reference description of a routine states that it sets the **errno** variable, you can use **errno** in two ways:

1. Compare **errno** to the values defined in the header file ERRNO.H.
2. Handle **errno** with the **perror** or **strerror** library routines. The **perror** routine prints a system error message to the standard error (**stderr**). The **strerror** routine stores the same information in a string for later use.

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 avoid confusion, you should always test the return value 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.

Library math routines set **errno** by calling the **matherr** or **_matherrl** library routines, which are described in the reference section. If you wish to handle math errors differently from these routines, you can write your own routine and name it **matherr** or **_matherrl**. Your routine must follow the rules listed in the **matherr** reference description.

The **ferror** library routine allows you to check for errors in stream input/output operations. This routine checks if an error indicator has been set for a given stream. Closing or rewinding the stream automatically clears the error indicator. You can also reset the error indicator by calling the **clearerr** library routine.

The **feof** library routine 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** routine or when a **read** operation returns 0 as the number of bytes read.

The **_grstatus** library routine allows you to check for errors after calling certain graphics library operations. See the reference page on the **_grstatus** function for details.

1.7 Operating-System Considerations

The library routines listed in this section behave differently under different operating system versions. For more information on an individual routine, see the description of that routine in the reference section.

Routine	Restrictions
locking **sopen** **_fsopen**	These routines are effective only in OS/2 and in DOS versions 3.0 and later.
dosexterr	The **dosexterr** routine provides error handling for system call 0x59 (get extended error) in DOS versions 3.0 and later.
dup **dup2**	The **dup** and **dup2** routines can cause unexpected results in DOS versions earlier than 3.0. If you use **dup** or **dup2** to create a duplicate file handle for **stdin, stdout, stderr, stdaux,** or **stdprn**, calling the **close** function with one handle causes errors in later I/O operations that use the other handle. This anomaly does not occur in OS/2 or in DOS versions 3.0 and later.
exec **spawn**	When using the **exec** and **spawn** families of functions under DOS versions earlier than 3.0, the value of the *arg0* argument (or *argv*[0] to the child process) is not available to the user; a null string (" ") is stored in that position instead. In OS/2, the *arg0* argument contains the command name; in DOS versions 3.0 and later, it contains the complete command path.

Microsoft C defines global variables that indicate the version of the current operating system. You can use these to determine the operating-system version in which a program is executing. See Chapter 3, "Global Variables and Standard Types," for more information.

1.8 Floating-Point Support

Microsoft math library routines require floating-point support to perform calculations with real numbers (numbers that can contain fractions). This support can be provided by the floating-point libraries that accompany your compiler software or by an 8087, 80287, or 80387 coprocessor. The names of the functions that require floating-point support are listed below:

acos	**cos**	**fmodl**	**powl**
acosl	**cosl**	**fmsbintoieee**	**sin**
asin	**cosh**	**_fpreset**	**sinl**
asinl	**coshl**	**frexp**	**sinh**
atan	**dieeetomsbin**	**frexpl**	**sinhl**
atanl	**difftime**	**gcvt**	**sqrt**
atan2	**dmsbintoieee**	**hypot**	**sqrtl**
atan2l	**ecvt**	**hypotl**	**_status87**
atof	**exp**	**ldexp**	**strtod**
_atold	**expl**	**ldexpl**	**_strtold**
bessel	**fabs**	**log**	**tan**
cabs	**fabsl**	**logl**	**tanl**
cabsl	**fcvt**	**log10**	**tanh**
ceil	**fieeetomsbin**	**log10l**	**tanhl**
ceill	**floor**	**modf**	
_clear87	**floorl**	**modfl**	
_control87	**fmod**	**pow**	

Note that the **bessel** routine does not correspond to a single function, but to twelve functions named **j0**, **j1**, **jn**, **y0**, **y1**, **yn**, **_j0l**, **_j1l**, **_jnl**, **_y0l**, **_y1l**, and **_ynl**. Also note that the **_clear87** and **_control87** functions are not available with the /FPa compiler option.

Also requiring floating-point support is the **printf** family of functions (**cprintf**, **fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, and **vsprintf**). These functions require 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.

When you use a floating-point type specifier in the format string for a **printf** or **scanf** call, make sure you specify floating-point values or pointers to floating-point values in the argument list. These must correspond to any floating-point type specifiers in the format string. The presence of floating-point arguments allows the compiler to detect that floating-point support code is required. If a floating-point type specifier 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 causes an error */
 {
  long f = 10L;
  printf("%f", f);
 }
```

In the preceding example, the functions for floating-point support are not loaded because

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

As a result, the following error occurs:

```
Floating point not loaded
```

Here is a corrected version of the above call to **printf** in which the long integer value is cast to **double:**

```
main( )  /* This example works correctly */
 {
  long f = 10L;
  printf("%f", (double) f);
 }
```

1.9 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**. However, generally, you cannot pass **_huge** data items as arguments to C library functions. 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	**_fmemchr**	**_fmemmove**	**lfind**
fread	**_fmemcmp**	**_fmemset**	**lsearch**
fwrite	**_fmemcpy**	**halloc**	**memccpy**
_fmemccpy	**_fmemicmp**	**hfree**	**memchr**

With this set of functions, you can read from, write to, search, sort, copy, initialize, compare, or dynamically allocate and free **_huge** arrays; the **_huge** array can be passed without difficulty to any of these functions in a compact-, large-, or huge-model program. The model-independent routines in the above list (those beginning with **_f**) are available in all memory models.

The **memset**, **memcpy**, and **memcmp** library routines are available in two versions: as C functions and as intrinsic (in-line) code. The function versions of these routines support huge pointers in compact and large memory models, but the intrinsic versions do not support huge pointers. (The function version of such routines generates a call to a library function, whereas the intrinsic version inserts in-line code into your program. Your compiler documentation explains how to select the intrinsic versions of library routines.)

CHAPTER

2

Run-Time Routines by Category

Microsoft C library routines handle various kinds of tasks. If you know the type of task you need done, but don't know exactly which routine to use, the categorized lists of routines in this chapter can help.

The descriptions here are intended only to give you a brief overview of the capabilities of the run-time library. For a complete description of the behavior, syntax, and use of each routine, see Part 2, "Run-Time Functions."

The main categories of library routines are

- Buffer manipulation
- Character classification and conversion
- Data conversion
- Directory control
- File handling
- Graphics
- Input and output
- Internationalization
- Math
- Memory allocation
- Process and environment control
- Searching and sorting
- String manipulation
- System calls
- Time
- Variable-length argument lists

2.1 Buffer Manipulation

The buffer-manipulation routines are useful for working with areas of memory on a character-by-character basis. A "buffer" is an array of characters, similar to a character string. However, unlike strings, buffers are not usually terminated with a null character ('\0'). Therefore, the buffer-manipulation routines always take a *length* or *count* argument. Function declarations for the buffer-manipulation routines are given in the include files MEMORY.H and STRING.H, with an exception being the **swab** function, which appears in STDLIB.H.

Routines beginning with **_f** are model independent; the **_f** stands for **far.** These routines are useful in writing mixed-model programs because they can be called from any program, regardless of the memory model being used.

Routine	Use
memccpy, _fmemccpy	Copy characters from one buffer to another until a given character or a given number of characters has been copied
memchr, _fmemchr	Return a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer
memcmp, _fmemcmp	Compare a specified number of characters from two buffers
memcpy, _fmemcpy	Copy a specified number of characters from one buffer to another
memicmp, _fmemicmp	Compare a specified number of characters from two buffers without regard to the case of the letters (uppercase and lowercase treated as equivalent)
memmove, _fmemmove	Copy a specified number of characters from one buffer to another
memset, _fmemset	Use a given character to initialize a specified number of bytes in the buffer
swab	Swaps bytes of data and stores them at the specified location

When the source and target areas overlap, only the **memmove** and **_fmemmove** functions are guaranteed to copy the full source properly. (The **memcpy** and **_fmemcpy** routines do not always copy the full source in such cases.)

2.2 Character Classification and Conversion

The character classification and conversion routines allow you to test individual characters in a variety of ways and to convert between uppercase and lowercase characters.

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 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))
```

All of these routines are implemented in two versions: as functions and as macros. The function prototypes and macro definitions appear in CTYPE.H. Section 1.4, "Choosing Between Functions and Macros," explains how to choose the appropriate version. The **toupper** and **tolower** functions are also declared in the STDLIB.H header file.

2.3 Data Conversion

The data-conversion routines convert numbers to strings of ASCII characters and vice versa. These routines are implemented as functions, all of which 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.

Routine	Use
abs	Finds absolute value of integer
atof	Converts string to **float**
atoi	Converts string to **int**
atol	Converts string to **long**
_atold	Converts string to **long double**
ecvt	Converts **double** to string
fcvt	Converts **double** to string
gcvt	Converts **double** to string
itoa	Converts **int** to string
labs	Finds absolute value of **long** integer
ltoa	Converts **long** to string
strtod	Converts string to **double**

strtol	Converts string to a **long** integer
_strtold	Converts string to **long double**
strtoul	Converts string to an **unsigned long** integer
ultoa	Converts **unsigned long** to string

2.4 Directory Control

The directory-control routines let a program access, modify, and obtain information about the directory structure. These routines are functions and are declared in DIRECT.H.

Routine	Use
chdir	Changes current working directory
_chdrive	Changes current drive
getcwd	Gets current working directory
_getdcwd	Gets current working directory for the specified drive
_getdrive	Gets the current disk drive
mkdir	Makes a new directory
rmdir	Removes a directory
_searchenv	Searches for a given file on specified paths

2.5 File Handling

The file-handling routines let you create, manipulate, and delete files. They also set and check file-access permissions.

File-handling routines work on a file designated by a path name or by a "file handle," an integer assigned by the operating system that identifies an open file. These routines modify or give information about the designated file. Most of them are declared in the include file IO.H, with the exceptions being the **fstat** and **stat** functions (declared in SYS\STAT.H), the **_fullpath** routine (declared in DIRECT.H), and the **remove** and **rename** functions (also declared in STDIO.H).

Routine	Use
access	Checks file-permission setting
chmod	Changes file-permission setting
chsize	Changes file size
filelength	Gets file length
fstat	Gets file-status information on handle
_fullpath	Makes an absolute path name from a relative path name
isatty	Checks for character device
locking	Locks areas of file (available with OS/2 and DOS versions 3.0 and later)
_makepath	Merges path-name components into a single, full path name
mktemp	Creates unique file name
remove	Deletes file
rename	Renames file
setmode	Sets file-translation mode
_splitpath	Splits a path name into component pieces
stat	Gets file-status information on named file
umask	Sets default-permission mask
unlink	Deletes file

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

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

The **mktemp** and **umask** routines have functions that are slightly different from the other routines. The **mktemp** routine creates a unique file name, and the programmer 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 can override the permission setting given in the **open** or **creat** call for the new file.

2.6 Graphics

Microsoft C graphics routines offer a wide variety of graphics functions, low-level graphics primitives, font functions, and presentation graphics (displays such as graphs and pie charts).

Graphics functions are supplied in two libraries that must be explicitly linked with your program. The GRAPHICS.LIB library provides support for low-level graphics and character-font routines. The library PGCHART.LIB supports presentation-graphics routines.

2.6.1 Low-Level Graphics and Character-Font Functions

The low-level graphics and font functions are declared in the include file GRAPH.H.

The library can be divided into the eight categories listed below, which correspond to the different tasks involved in creating and manipulating graphic objects.

Most graphics routines work only in DOS. Two categories of routines ("configuring mode and environment" and "creating text output") work in OS/2 as well as DOS.

Category	Task
Configuring mode and environment (OS/2 and DOS)	Select the proper display mode for the hardware and establish memory areas for writing and displaying of images
Setting coordinates	Specify the logical origin and the active display area within the screen
Setting low-level graphics palettes	Specify a palette mapping for low-level graphics routines
Setting attributes	Specify background and foreground colors, fill masks, and line styles for low-level graphics routines
Creating graphics output	Draw and fill figures
Creating text output (OS/2 and DOS)	Write text on the screen
Transferring images	Store images in memory and retrieve them
Displaying fonts	Display text in character fonts compatible with Microsoft Windows™

The following sections explain each of these categories.

2.6.1.1 Configuring Mode and Environment

Routines that configure the mode and environment establish the graphics or text mode of operation, determine the current graphics environment, and control the display of the cursor.

All of the routines listed in this section are available in OS/2 as well as DOS.

Routine	Use
_clearscreen	Erases the screen and fills it with the current background color
_getactivepage	Gets the current active page number
_getbkcolor	Returns the current background color
_getvideoconfig	Obtains status of current graphics environment
_getvisualpage	Gets the current visual page number
_grstatus	Returns the status of the most recent graphics function call
_setactivepage	Sets memory area for the active page for writing images
_setbkcolor	Sets the current background color
_settextrows	Sets the number of text rows
_setvideomode	Selects an operating mode for the display screen
_setvideomoderows	Sets the video mode and the number of rows for text operations
_setvisualpage	Sets memory area for the current visual page

2.6.1.2 Setting Coordinates

The "set coordinates" routines set the current text or graphics position and convert pixel coordinates between the various graphic coordinate systems.

The Microsoft C graphics functions recognize three sets of coordinates:

1. Fixed physical coordinates
2. View coordinates defined by the application
3. Window coordinates that can include floating-point values

The functions in this category establish window and view coordinate systems and translate between physical, view, and window coordinate systems.

Routine	Use
_getcurrentposition	Determines current position in view coordinates
_getcurrentposition_w	Determines current position in window coordinates
_getphyscoord	Converts view coordinates to physical coordinates
_getviewcoord	Converts physical coordinates to view coordinates
_getviewcoord_w	Converts window coordinates to view coordinates
_getviewcoord_wxy	Converts window coordinates in **_wxycoord** structure to view coordinates
_getwindowcoord	Converts view coordinates to window coordinates
_setcliprgn	Limits graphic output to a region of the screen
_setvieworg	Positions the view-coordinate origin
_setviewport	Limits graphics output to a region of the screen and positions the view-coordinate origin to the upper-left corner of that region
_setwindow	Defines a floating-point window coordinate system

The default view coordinate system is identical to the physical screen coordinate system. 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.

The physical horizontal and vertical dimensions depend on 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 **_getvideoconfig** routine is listed in the previous section.)

The **_setvieworg** function allows you to move the viewport origin to a new position relative to the physical screen.

Routines that refer to coordinates on the physical screen or viewport require integer values. However, in real-world graphing applications, you might wish to use floating-point values, such as stock prices or average rainfall. The window coordinate system allows you to display graphics using floating-point values instead of integers.

The **_getcurrentposition** and **_getcurrentposition_w** routines allow you to determine the location of the current graphics-output point.

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

The physical coordinates of any view-coordinate point can be determined with the **_getphyscoord** function, and the view coordinates of any physical point can be determined with the **_getviewcoord** function.

The view coordinates of any window coordinate can be determined with the **_getviewcoord_w** and **_getviewcoord_wxy** functions. The window coordinates of any view coordinate can be determined with the **_getwindowcoord** function.

The **_setwindow** function defines the current viewport as a real-coordinate window bound by the specified floating-point values.

2.6.1.3 Setting Low-Level Graphics Palettes

Use the low-level palette routines to select or remap color palettes.

Routine	Use
_remapallpalette	Changes all color indexes in the current palette
_remappalette	Changes a single color index in the current palette
_selectpalette	Selects a predefined palette

Some video modes support a "color palette," which is a table of the color values that can be displayed together on the screen at any given time. A "color value" is a **long** integer representing a color that can be displayed on your system.

In CGA color graphics modes, you can use the **_selectpalette** routine to choose one of several predefined palettes.

On EGA and VGA video systems, you can "remap" (change) the palette using the **_remappalette** or **_remapallpalette** routines. For instance, the EGA **_ERESCOLOR** mode offers a total of 64 color values, of which 16 can be displayed at a time. In this mode, the palette contains 16 "color indices," or slots to which you can assign color values.

The **_remappalette** routine changes a single color index to a specified color value. The **_remapallpalette** routine changes all of the available palette entries simultaneously.

2.6.1.4 Setting Attributes

The low-level output functions that draw lines, arcs, ellipses, and other basic figures do not specify color or line-style information. Instead, the low-level graphics functions rely on a set of attributes that are set independently by the following functions:

Routine	Use
_getarcinfo	Determines the endpoints in viewport coordinates of the most recently drawn arc or pie
_getcolor	Gets the current color
_getfillmask	Gets the current fill mask
_getlinestyle	Gets the current line-style mask
_getwritemode	Gets the current logical write mode
_setcolor	Sets the current color
_setfillmask	Sets the current fill mask
_setlinestyle	Sets the current line-style mask
_setwritemode	Sets logical write mode for line drawing

The **_getcolor** and **_setcolor** functions get or set the current color index for graphics and font output. The **_getbkcolor** and **_setbkcolor** functions get or set the current background color.

The **_getfillmask** and **_setfillmask** functions get or set the current fill mask. 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, as the mask is transparent to that pixel. If a bit is 1, the pixel is assigned the current color value. The template is repeated as necessary over the entire fill area.

The **_getlinestyle** and **_setlinestyle** functions get or set the current line style. The line style is determined by a 16-bit template buffer with each bit corresponding to a pixel. If a bit is 1, the pixel is set to the current color. If a bit is 0, the pixel is not changed. The template is repeated for the length of the line.

The **_getwritemode** and **_setwritemode** functions get or set the logical write mode for straight line drawing. The default mode, **_GPSET**, causes lines to be drawn in the current graphics color. Other modes combine the current graphics color and the original screen image using various logical operations.

2.6.1.5 Creating Graphics Output

The graphics output functions use a set of specified coordinates and draw various figures. They use the current or default attributes for line-style mask, fill mask, write mode, background color, and foreground color.

The name of each function announces its task or the figure it draws, as the following list indicates:

Routine	Use
_arc, _arc_w, _arc_wxy	Draw an arc
_ellipse, _ellipse_w, _ellipse_wxy	Draw an ellipse or circle
_floodfill, _floodfill_w	Flood-fill an area of the screen with the current color
_getcurrentposition, _getcurrentposition_w	Obtain the current graphic-output position used by **_lineto** and **_outgtext**
_getpixel, _getpixel_w	Obtain a pixel's color
_lineto, _lineto_w	Draw a line from the current graphic output position to a specified point
_moveto, _moveto_w	Move the current graphic-output position to a specified point
_pie, _pie_w, _pie_wxy	Draw a pie-slice-shaped figure
_polygon, _polygon_w, _polygon_wxy	Draw or scan-fill a polygon
_rectangle, _rectangle_w, _rectangle_wxy	Draw or scan-fill a rectangle
_setpixel, _setpixel_w	Set a pixel's color

Most of these routines are available in several forms, which are indicated by their names. Output functions without a suffix use the view coordinate system. Functions that end with _w take **double** values as arguments and use the window coordinate system. Functions that end with **_wxy** use **_wxycoord** structures to define the coordinates and use the window coordinate system.

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.

2.6.1.6 Creating Text Output

The next group of routines provides text output in both graphics and text modes. Unlike the standard console I/O library routines, these functions recognize text-window boundaries and use the current text color.

All of the routines listed in this section work in OS/2 as well as DOS.

Routine	Use
_displaycursor	Sets the cursor on or off upon exit from a graphics routine
_gettextcolor	Obtains the current text color
_gettextcursor	Returns the current cursor attribute (text modes only)
_gettextposition	Obtains the current text-output position
_gettextwindow	Gets the current text window boundaries
_outmem	Prints text of a specified length from a memory buffer
_outtext	Outputs a text string to the screen at the current text position
_scrolltextwindow	Scrolls the current text window up or down
_settextcolor	Sets the current text color
_settextcursor	Sets the current cursor attribute (text modes only)
_settextposition	Relocates the current text position
_settextwindow	Defines the current text-display window
_wrapon	Enables or disables line wrap

The **_outtext** and **_outmem** routines provide no formatting. If you want to output integer or floating-point values, you must convert the values into a string variable (using the **sprintf** function) before calling these routines.

The **_outtext** routine recognizes the **\n** (newline character) and **\r** (carriage return) sequences. The **_outmem** routine treats these sequences as printable graphics characters.

2.6.1.7 Transferring Images

The functions in this category transfer screen images between memory and the display, using a buffer allocated by the application, or determine the size in bytes of the buffer needed to store a given image.

The functions that end with **_w** or **_wxy** use window coordinates; the other functions in this set use view coordinates.

Routine	Use
_getimage, _getimage_w, _getimage_wxy	Store a screen image in memory
_imagesize, _imagesize_w, _imagesize_wxy	Return the size (in bytes) of the buffer needed to store the image
_putimage, _putimage_w	Retrieve an image from memory and display it

In some cases, the buffer needed to store an image with the **_getimage** functions must be larger than 64K (65,535) bytes. Use the **halloc** routine to allocate a buffer larger than 64K.

2.6.1.8 Displaying Fonts

The functions listed in this section control the display of font-based characters on the screen.

Routine	Use
_getfontinfo	Obtains the current font characteristics
_getgtextextent	Determines the width in pixels of specified text in the current font
_getgtextvector	Gets orientation of font text output
_outgtext	Outputs text in the current font to the screen at the specified pixel position
_registerfonts	Initializes font library
_setfont	Finds a single font that matches a specified set of characteristics and makes this font the current font for use by the **_outgtext** function
_setgtextvector	Sets the current orientation for font text output
_unregisterfonts	Frees memory allocated by **_registerfonts**

2.6.2 Presentation-Graphics Functions

The presentation-graphics functions are declared in the PGCHART.H include file. The library can be divided into the three categories listed below, corresponding to the different tasks involved in creating and manipulating graphic objects:

Category	Task
Displaying presentation graphics	Initialize video structures for presentation graphics and establishes the default chart type. Display presentation-graphics chart: bar, column, pie, scatter, or line chart.
Analyzing presentation-graphics data	Analyze data (does not display chart).
Manipulating presentation-graphics structures	Modify basic chart structures (e.g., palettes, cross-hatching styles).

2.6.2.1 Displaying Presentation Graphics

The functions listed in this section initialize the presentation-graphics library and display the specified graph type.

Because the **_pg_initchart** routine initializes the presentation-graphics library, it must be called before any other function in the presentation-graphics library. The **_pg_defaultchart** function initializes the variables in the chart environment.

The other routines in this category display the specified graph. The single-series versions plot one set of data, and the multiseries versions (those ending with an **ms** suffix) plot several sets of data in the same chart style.

Presentation-graphics programs can display text in different font sizes by taking advantage of font-based characters (see Section 2.6.1.8, "Displaying Fonts.") Call the **_registerfonts** and **_setfont** routines to select a font before calling the **_pginitchart** routine. Subsequent charts use the selected font. You can later call the **_unregisterfonts** routine to restore the default character font and free the memory previously allocated for fonts.

Routine	Use
_pg_chart	Displays a single-series bar, column, or line chart
_pg_chartms	Displays a multiseries bar, column, or line chart
_pg_chartpie	Displays a pie chart
_pg_chartscatter	Displays a scatter diagram for a single series of data
_pg_chartscatterms	Displays a scatter diagram for more than one series of data
_pg_defaultchart	Initializes all necessary variables in the chart environment for a specified chart type
_pg_initchart	Initializes the presentation-graphics library

2.6.2.2 Analyzing Presentation-Graphics Charts

These routines calculate default values for the specified graph type but do not display the chart. The single-series versions analyze one set of data, and the multiseries versions analyze several sets of data in the same chart style.

Routine	Use
_pg_analyzechart	Analyzes a single series of data for a bar, column, or line chart
_pg_analyzechartms	Analyzes a multiseries of data for a bar, column, or line chart
_pg_analyzepie	Analyzes data for a pie chart
_pg_analyzescatter	Analyzes a single series of data for a scatter diagram
_pg_analyzescatterms	Analyzes a multiseries of data for a scatter diagram

2.6.2.3 Manipulating Presentation-Graphics Structures

These functions control low-level aspects of the presentation-graphics package.

Routine	Use
_pg_hlabelchart	Writes text horizontally on the screen
_pg_vlabelchart	Writes text vertically on the screen
_pg_getpalette	Retrieves current colors, line styles, fill patterns, and plot characters for all presentation-graphics palettes
_pg_setpalette	Sets current colors, line styles, fill patterns, and plot characters for all presentation-graphics palettes
_pg_resetpalette	Sets current colors, line styles, fill patterns, and plot characters to the default values for the current screen mode
_pg_getstyleset	Retrieves the contents of the current styleset
_pg_setstyleset	Sets the contents of the current styleset
_pg_resetstyleset	Resets the contents of the current styleset to the default value for the current screen mode
_pg_getchardef	Retrieves the current 8-by-8-pixel bit map for a specified character
_pg_setchardef	Sets the 8-by-8-pixel bit map for a specified character

2.7 Input and Output

The input and output (I/O) 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 items are treated as sequences of bytes. The following three types of I/O functions are available:

1. Stream
2. Low-level
3. Console and port

The "stream" I/O functions treat data 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. Stream I/O also provides buffering, which can significantly improve performance.

The "low-level" I/O routines do not perform buffering and formatting. Instead, 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.

The "console and port" I/O routines allow you to read or write directly to a console (keyboard and screen) or an I/O port (such as a printer port). The port I/O routines simply read and write data in bytes. With console I/O routines, some additional options are available, such as detecting 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 C library also provides a number of direct DOS I/O system call routines. These are described in Section 2.14, "System Calls."

File I/O operations can be performed in two modes: text or binary. The following section describes these modes and their use.

WARNING *Because stream routines are buffered and low-level routines are not, the two types of routines are generally incompatible. You should use either stream or low-level routines consistently for processing a given file.*

2.7.1 Text and Binary Modes

Many C programs use data files for input and output. Under DOS and OS/2, data files are normally processed in text mode. In this mode, each carriage-return–line-feed (CR-LF) combination is translated into a single line-feed character during input. During output, each line-feed character is translated into a CR-LF combination.

Sometimes you may want to process a file without making those translations. In these cases you use binary mode, which suppresses CR-LF translations.

You can control the file translation mode 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** routine opens a file in binary mode when you specify the letter **b** in the access-mode string for the file. The **open** routine opens a file in binary mode when you specify the **O_BINARY** flag in the *oflag* argument. For more information about **fopen** and **open**, see the reference description of each routine.
- 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, which is normally text. If you set **_fmode** to **O_BINARY**, the default mode is binary except for **stdaux** and **stdprn**, which are opened in binary mode by default.

You can change the value of **_fmode** in two ways:

1. Link with the file BINMODE.OBJ (supplied with Microsoft C). This changes the initial setting of **_fmode** to the **O_BINARY** flag, causing all files except **stdin**, **stdout**, and **stderr** to be opened in binary mode.
2. Change the value of **_fmode** directly by setting it to the **O_BINARY** flag in your program. This has the same effect as linking with BINMODE.OBJ.

You can still override the default mode (now binary) for a particular file by opening it in text mode. Specify the letter **t** when using **fopen**, or specify the **O_TEXT** flag when using **open**.

By default, the **stdin**, **stdout**, and **stderr** files are opened in text mode, and the **stdaux** and **stdprn** files are opened in binary mode. The **setmode** routine allows you to change these defaults or change the mode of a file after it has been opened. See the reference description of **setmode** for details.

2.7.2 Stream Routines

Stream I/O functions handle data as a continuous stream of characters. 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.

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 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.

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.

Routine	Use
clearerr	Clears the error indicator for a stream
fclose	Closes a stream
fcloseall	Closes all open streams
fdopen	Associates a stream with an open file 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 the 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 to a new file
fscanf	Reads formatted data from a stream
fseek	Moves file position to a given location
fsetpos	Sets the position indicator of a stream
_fsopen	Opens a stream with file sharing
ftell	Gets current file position
fwrite	Writes unformatted data items to a stream
getc	Reads a character from a stream
getchar	Reads a character from **stdin**
gets	Reads a line from **stdin**
getw	Reads a binary **int** item from a stream
printf	Writes formatted data to **stdout**
putc	Writes a character to a stream
putchar	Writes a character to **stdout**
puts	Writes a line to a stream
putw	Writes a binary **int** item to a stream
rewind	Moves file position 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 a string
sscanf	Reads formatted data from a 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

2.7.2.1 Opening a Stream

A stream must be opened using the **fdopen**, **fopen**, **freopen**, or **_fsopen** 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 in either text or binary mode.

The **fdopen, fopen, freopen**, and **_fsopen** functions return a **FILE** pointer. You normally assign the pointer value to a variable and use the variable to refer to the opened stream. For instance, if your program contains the lines

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

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

2.7.2.2 Using Predefined Stream Pointers

When a program begins execution, the C start-up code automatically opens several streams: standard input, standard output, and standard error. By default, the standard input, standard output, and standard error streams are directed to the console (keyboard and screen). This means that when 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.

Under DOS, two additional streams are opened: standard auxiliary and standard print. (These streams are not available in OS/2.) The assignment of standard auxiliary and standard print depends on the machine configuration. These streams usually refer to the first serial port and a printer port, but those ports may not be available on some systems. Be sure to check your machine configuration before using these streams.

You can refer to the standard streams with the following predefined stream pointers:

Pointer	Stream
stdin	Standard input
stdout	Standard output
stderr	Standard error
stdaux	Standard auxiliary (DOS only)
stdprn	Standard print (DOS only)

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.

DOS and OS/2 allow you to redirect a program's standard input and standard output at the operating-system command level. OS/2 also allows you to redirect a program's standard error. See your operating system user's manual for a complete discussion of redirection.

Within your program, you can use **freopen** to redirect **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** so that it refers to a disk file or to a device. See the reference description of **freopen** for more details.

2.7.2.3 Controlling Stream Buffering

As mentioned earlier, stream routines can use in-memory buffers to speed I/O operations. Files opened using the stream routines are buffered by default, except for **stdaux** and **stdprn**, which are normally unbuffered. The **stdout** and **stderr** streams are flushed whenever they are full or (if you are writing to a character device) after each library call.

By using the **setbuf** or **setvbuf** function, 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 you, but buffers allocated with **setbuf** or **setvbuf** refer to arrays in your program and can be manipulated. Buffers can be any size up to 32,767 bytes. This size is set by the manifest constant **BUFSIZ** in STDIO.H if you use **seftbuf**; if you use **setvbuf**, you can set the size of the buffer yourself. (See the descriptions of **setbuf** and **setvbuf** in the reference section for more details.)

NOTE *These routines affect only buffers created by C library routines. They have no effect on buffers created by the operating system.*

2.7.2.4 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 your program is finished with it, as the number of streams that can be open at a given time is limited.

2.7.2.5 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.

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 **fseek** and **fsetpos** functions allow you to position the file pointer anywhere in a file. The next operation occurs at the position you specified. The **rewind** routine positions the file pointer at the beginning of the file. Use the **ftell** or **fgetpos** routine to determine the current position of the file pointer.

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, fseek, fsetpos,** or **rewind** is called.

Streams associated with a character-oriented 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 character-oriented device.

2.7.2.6 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.**

2.7.3 Low-Level Routines

Low-level input and output calls do not buffer or format data. Declarations for the low-level functions are given in the include files IO.H, FCNTL.H, SYS\TYPES.H, and SYS\STAT.H. Unlike the stream functions, low-level 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.

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
umask	Sets default file-permission mask
write	Writes data to a file

2.7.3.1 Opening a File

You must open a file before performing I/O functions on it. The **open** function opens a file; it can also create the file when opening it. In OS/2 and DOS versions 3.0 and later, you can use **sopen** to open a file with file-sharing attributes. The **creat** function can create and open a file.

The file can be opened for reading, writing, or both, and opened in either text or binary mode (see Section 2.7.1, "Text and Binary Modes"). 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 description for the **open** function.

These functions return a file handle, which is normally assigned to an integer variable. You use the variable to refer to the opened file.

2.7.3.2 Reading and Writing Data

Use the **read** and **write** routines to read and write to files. These operations begin at the current position in the file. The current position is updated each time a read or write operation occurs.

The **lseek** function allows you to place the file pointer anywhere in the file. The next operation occurs at the position you specified. The **tell** function indicates the current position of the file pointer. The **eof** routine tests for the end of the file.

Low-level I/O routines set the **errno** variable when an error occurs. Chapter 3, "Global Variables and Standard Types," describes **errno**.

Character-oriented 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.

2.7.3.3 Closing Files

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

2.7.3.4 Using Predefined Handles

When a program begins execution, three files are automatically opened: standard input, standard output, and standard error. In DOS, two additional files are opened: standard auxiliary and standard print. (These files are not available in OS/2.)

Low-level routines can access these files using the following predefined handles:

Stream	Handle
stdin	0
stdout	1
stderr	2
stdaux (DOS only)	3
stdprn (DOS only)	4

You can use these file handles without previously opening the files. The files are opened and the handles are assigned when the program starts.

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.

In DOS and OS/2, you can redirect the standard input and standard output at the operating-system command level. OS/2 also allows you to redirect the standard error. See your operating system user's manual for a complete discussion of redirection.

2.7.4 Console and Port I/O

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.

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

NOTE *Programs that need only run under DOS can also use a number of direct DOS I/O system calls (* ***_dos_open, _dos_read, _dos_close,*** *etc.) These are described in detail in Section 2.14, "System Calls."*

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 **inpw** and **outpw** routines read and 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.

NOTE *The console I/O routines are not compatible with stream or low-level library routines and should not be used with them.*

2.8 Internationalization

Internationalization routines are useful for creating different versions of a program for international markets. These routines are declared in the header file LOCALE.H, except for **strftime**, which is declared in TIME.H.

Routine	Use
localeconv	Sets a structure with appropriate values for formatting numeric quantities
setlocale	Selects the appropriate locale for the program
strcoll	Compares strings using locale-specific information
strftime	Formats a date and time string
strxfrm	Transforms a string based on locale-specific information

2.9 Math

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 1.8, "Floating-Point Support").

The math library provides two versions of some routines. The first version of the routine supports **double** arguments and return values. The second version supports an 80-bit data type, allowing the routine to take **long double** arguments and return a **long double** value. The second version usually has the same name with the suffix **l**. For instance, the **acos** routine supports **double** arguments and return values, while **acosl** supports **long double** arguments and return values.

Routines which support **long double** values are not available when you compile with the /Fpa (alternate math) compiler option. The same is true of the **_clear 87**, **_control87**, and **_status87** routines.

Most math declarations are in the include file MATH.H. However, the **_clear87**, **_control87**, **_fpreset**, and **_status87** routines are defined in FLOAT.H, the **abs** and **labs** functions are defined in MATH.H and STDLIB.H, and the **div** and **ldiv** routines are declared in STDLIB.H.

Routine	Use
acos, acosl	Calculate the arccosine
asin, asinl	Calculate the arcsine
atan, atanl	Calculate the arctangent
atan2, atan2l	Calculate the arctangent
bessel	Calculates Bessel functions
cabs, cabsl	Find the absolute value of a complex number
ceil, ceill	Find the integer ceiling
_clear87	Gets and clears the floating-point status word
_control87	Gets the old floating-point control word and sets a new control-word value
cos, cosl	Calculate the cosine
cosh, coshl	Calculate the hyperbolic cosine
dieeetomsbin	Converts IEEE double-precision number to Microsoft (MS) binary format
div	Divides one integer by another, returning the quotient and remainder
dmsbintoieee	Converts Microsoft binary double-precision number to IEEE format
exp, expl	Calculate the exponential function
fabs, fabsl	Find the absolute value

fieeetomsbin	Converts IEEE single-precision number to Microsoft binary format
floor, floorl	Find the largest integer less than or equal to the argument
fmod, fmodl	Find the floating-point remainder
fmsbintoieee	Converts Microsoft binary single-precision number to IEEE format
_fpreset	Reinitializes the floating-point-math package
frexp, frexpl	Calculate an exponential value
hypot, hypotl	Calculate the hypotenuse of right triangle
ldexp, ldexpl	Calculate the product of the argument and 2^{exp}
ldiv	Divides one **long** integer by another, returning the quotient and remainder
log, logl	Calculate the natural logarithm
log10, log10l	Calculate the base-10 logarithm
_lrotl, _lrotr	Shift an **unsigned long int** item left (**_lrotl**) or right (**_lrotr**)
matherr, _matherrl	Handle math errors
max, min	Return the larger or smaller of two values
modf, modfl	Break down the argument into integer and fractional parts
pow, powl	Calculate a value raised to a power
rand	Gets a pseudorandom number
_rotl, _rotr	Shift an **unsigned int** item left (**_rotl**) or right (**_rotr**)
sin, sinl	Calculate the sine
sinh, sinhl	Calculate the hyperbolic sine
sqrt, sqrtl	Find the square root
srand	Initializes a pseudorandom series
_status87	Gets the floating-point status word
tan, tanl	Calculate the tangent
tanh, tanhl	Calculate the hyperbolic tangent

The **bessel** routine does not correspond to a single function, but to twelve functions named **j0, j1, jn, y0, y1, yn, _j0l, _j1l, _jnl, _y0l, _y1l,** and **_ynl.**

The **matherr** and **_matherrl** routines are invoked by the math functions when errors occur. The **matherr** routine handles functions that return a **double** value and **_matherrl** handles routines that return a **long double.**

These routines are defined in the library, but you can redefine them for different error-handling. The user-defined function, if given, must follow the rules given in the reference description of **matherr** and **_matherrl.**

You are not required to supply a definition for the **matherr** routines. If no definition is present, the default error returns for each routine are used. The reference description of each routine describes that routine's error returns.

2.10 Memory Allocation

The memory-allocation routines allow you to allocate, free, and reallocate blocks of memory. Memory-allocation routines are declared in the include file MALLOC.H.

Routine	Use
alloca	Allocates a block of memory from the program's stack
_bfreeseg	Frees a based heap
_bheapseg	Allocates a based heap
calloc, _bcalloc, _fcalloc, _ncalloc	Allocate storage for an array
_expand, _bexpand, _fexpand, _nexpand	Expand or shrink a block of memory without moving its location
free, _bfree, _ffree, _nfree	Free an allocated block
_freect	Returns approximate number of items of given size that could be allocated in the near heap
halloc	Allocates storage for huge array
_heapadd, _bheapadd	Add memory to a heap
_heapchk, _bheapchk, _fheapchk, _nheapchk	Check a heap for consistency
_heapmin, _bheapmin, _fheapmin, _nheapmin	Release unused memory in a heap

_heapset, _bheapset, _fheapset, _nheapset	Fill free heap entries with a specified value
_heapwalk, _bheapwalk, _fheapwalk, _nheapwalk	Return information about each entry in a heap
hfree	Frees a block allocated by **halloc**
malloc, _bmalloc, _fmalloc, _nmalloc	Allocate a block of memory
_memavl	Returns approximate number of bytes available for allocation in the near heap
_memmax	Returns size of largest contiguous free block in the near heap
_msize, _bmsize, _fmsize, _nmsize	Return size of an allocated block
realloc, _brealloc, _frealloc, _nrealloc	Reallocate a block to a new size
stackavail	Returns size of stack space available for allocation with **alloca**

Some memory-management routines, such as **malloc**, are available in different versions that begin with **_b**, **_f**, or **_n**. These variations are described in the following section.

The **malloc** and **free** routines allocate and free memory space, respectively, while a program runs. The **malloc** routine allocates memory from the "heap," which is a pool of memory not otherwise used by your program. In tiny-, small-, and medium-model programs, the heap consists of unused memory in your program's default data segment. In compact-, large-, and huge-model programs, it is unused memory outside the default data segment.

The **malloc** and **free** routines satisfy the memory-allocation requirements of most programs. More specialized memory-management routines are discussed below.

The **realloc** and **_expand** routines can expand or shrink an allocated memory block. They behave differently in cases in which there is not enough room to expand the block in its current location. In this case, **realloc** moves the block as needed, but **_expand** does not.

The **calloc** routine allocates memory for an array and initializes every byte in the allocated block to 0.

The **halloc** routine is similar to **calloc**, except that it can allocate memory for a huge array (one that exceeds 64K in size). This routine is useful when you need a very large data object, or if you need to return allocated memory to the operating system for subsequent calls to the **spawn** family of functions.

2.10.1 Near and Far Heaps

As mentioned in the previous section, heap memory can reside inside or outside your program's default data segment, depending on what memory model your program uses. When it lies inside the default data segment, the heap is called the "near heap," since it can be accessed with near pointers. The "far heap" is memory that spans one or more segments outside the default data segment. The far heap can be accessed only with far pointers.

In various memory models, **malloc** automatically allocates memory from the near heap or far heap, as appropriate. The C library also includes near and far versions of **malloc**, **free**, and other memory-management routines, which allow you to specify the near and far heaps explicitly. These have the same names as standard memory routines, but are preceded by **_n** (for **near**) or **_f** (for **far**).

For instance, the **_nmalloc** routine always allocates memory from the near heap and returns a near pointer, no matter which memory model your program uses. Use **_nfree** to release memory allocated with **_nmalloc**.

Similarly, **_fmalloc** always allocates memory from the far heap and returns a far pointer, regardless of memory model. Use the **_ffree** routine to release memory allocated with **_fmalloc**.

2.10.2 Based Heaps

You can also allocate memory from a "based heap," which is a single segment that lies outside the default data segment. Based-heap routines generally use the same names as standard memory routines, but begin with **_b**. For instance, **_bmalloc** allocates a memory block from the based heap and **_bfree** frees the block.

Based heaps offer the following advantages:

- Localized data. Based heaps allow you to group related data in a single segment. This can simplify the management of related data. In OS/2, based heaps can also minimize the risk of general protection faults and improve performance.
- Faster pointer arithmetic. Although the based heap lies in the far data segment, pointers to its data items are the same size as near pointers. Thus, pointer arithmetic on items in a based heap is faster than pointer arithmetic on items in the far heap.

The **_bheapseg** routine allocates a based heap segment, from which you can then allocate blocks of memory. You can call **_bheapseg** more than once to allocate as many based-heap segments as needed (within the confines of available memory).

The **_bfreeseg** routine frees a based-heap segment. This routine frees every block in the based-heap segment, whether or not you previously freed the blocks individually.

NOTE *Near-, far-, and based-heap calls are not ANSI compatible and will make your program less portable.*

2.11 Process and Environment Control

The process-control routines allow you to start, stop, and manage processes from within a program. Environment-control routines allow you to get and change information about the operating-system environment.

A "process" is a program being executed by the operating system. It consists of the program's code and data, plus information about the process, such as the number of open files. Whenever you execute a program at the operating-system level, you start a process.

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 environment-control routines (**getenv** and **putenv**) are declared in STDLIB.H.

Routine	Use
abort	Aborts a process without flushing buffers or calling functions registered by **atexit** and **onexit**
assert	Tests for logic error
atexit	Schedules routines for execution at program termination
_beginthread	Creates an execution thread (OS/2 only)
_cexit	Performs the **exit** termination procedures (such as flushing buffers) and returns control to the calling program
_c_exit	Performs the **_exit** termination procedures and returns control to the calling program
cwait	Suspends the calling process until a specified child process terminates (OS/2 only)
_endthread	Terminates an execution thread (OS/2 only)

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	Calls functions registered by **atexit** and **onexit**, then flushes all buffers and closes all open files before terminating the process
_exit	Terminates process without processing **atexit** or **onexit** functions or flushing buffers
getenv	Gets the value of an environment variable
getpid	Gets process ID number
longjmp	Restores a saved stack environment
onexit	Schedules routines for execution at program termination
_pclose	Waits for a child command and closes a pipe on the associated stream
perror	Prints error message
_pipe	Creates a pipe
_popen	Creates a pipe and asynchronously executes a child copy of the command processor
putenv	Adds or changes the value of an environment variable
raise	Sends a signal to the calling process
setjmp	Saves a stack environment

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 operating system command
wait	Suspends the calling process until any of the caller's immediate child processes terminate (OS/2 only)

The **atexit** and **onexit** routines create a list of functions to be executed when the calling program terminates. The only difference between the two is that **atexit** is part of the ANSI standard. The **onexit** function is offered for compatibility with previous versions of Microsoft C.

The **_exit** routine terminates a process immediately, whereas **exit** terminates the process only after flushing buffers and calling any functions previously registered by **atexit** and **onexit**. The **_cexit** and **_c_exit** routines are identical to **exit** and **_exit**, respectively, except that they return control to the calling program without terminating the process.

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

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 (see Table 2.1). The differences among the forms involve the method of locating the file to be executed as the child process, the method for passing arguments to the child process, and the method of setting the environment.

Passing an argument list means that the arguments to the child process are listed separately 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. The argument-array method is useful when the number of arguments must be determined at run time.

Several process-control routines take advantage of the multitasking capability of OS/2. The **_beginthread** and **_endthread** routines create and terminate execution threads. The **cwait** and **wait** routines suspend the calling process until one child process terminates. The **_pipe**, **_popen**, and **_pclose** routines create and manipulate pipes, which link processes for sequential execution.

Table 2.1 Forms of the spawn and exec Routines

Routines	Locating the File	Argument-Passing Convention	Environment Settings
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

The **assert** macro is typically used to test for 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.

2.12 Searching and Sorting

Search and sort routines provide binary-search, linear-search, and quick-sort capabilities. They are all declared in SEARCH.H.

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

2.13 String Manipulation

The string functions are declared in the include file STRING.H. They allow you to compare strings, copy them, search for strings and characters, and perform various other operations.

Routines beginning with **_f** are model-independent versions of the corresponding routines and are useful in mixed-model programs. These routines can be called from any point in the program, regardless of which model is being used.

Routine	Use
strcat, _fstrcat	Append one string to another
strchr, _fstrchr	Find first occurrence of a given character in a string
strcmp, _fstrcmp	Compare two strings
strcpy, _fstrcpy	Copy one string to another
strcspn, _fstrcspn	Find first occurrence of a character from a given character set in a string
strdup, _fstrdup, _nstrdup	Duplicate a string
strerror	Maps an error number to a message string

_strerror	Maps a user-defined error message to a string
stricmp, _fstricmp	Compare two strings without regard to case
strlen, _fstrlen	Find length of string
strlwr, _fstrlwr	Convert string to lowercase
strncat, _fstrncat	Append characters of a string
strncmp, _fstrncmp	Compare characters of two strings
strncpy, _fstrncpy	Copy characters of one string to another
strnicmp, _fstrnicmp	Compare characters of two strings without regard to case
strnset, _fstrnset	Set characters of a string to a given character
strpbrk, _fstrpbrk	Find first occurrence of a character from one string in another
strrchr, _fstrrchr	Find last occurrence of a given character in string
strrev, _fstrrev	Reverse string
strset, _fstrset	Set all characters of a string to a given character
strspn, _fstrspn	Find first substring from a given character set in a string
strstr, _fstrstr	Find first occurrence of a given string in another string
strtok, _fstrtok	Find next token in a string
strupr, _fstrupr	Convert a string to uppercase

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 in Section 2.1.

2.14 System Calls

The following routines give access to IBM-PC BIOS interrupts and DOS system calls. Except for the **FP_OFF**, **FP_SEG**, and **segread** routines, these routines are for DOS application programs only; they do not work under OS/2.

2.14.1 BIOS Interface

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

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

NOTE *BIOS routines are hardware dependent. Some of them may not work as expected on machines whose hardware differs from the IBM PC.*

2.14.2 DOS Interface

These routines are implemented as functions and declared in DOS.H.

Routine	Use
bdos	Invokes 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 DOS system call 0x48
_dos_close	Closes a file, using DOS system call 0x3E
_dos_creat	Creates a new file and erases any existing file having the same name, using DOS system call 0x3C

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

_dos_settime	Sets the system time, using DOS system call 0x2D
_dos_setvect	Sets a new value for the specified interrupt vector, using DOS system call 0x25
_dos_write	Sends output to a file, using DOS system call 0x40
dosexterr	Obtains in-depth error information from DOS system call 0x59
_enable	Enables interrupts
FP_OFF	Returns offset portion of a far pointer (OS/2 and DOS)
FP_SEG	Returns segment portion of a far pointer (OS/2 and DOS)
_harderr	Establishes a hardware error handler
_hardresume	Returns to DOS after a hardware error
_hardretn	Returns to the application after a hardware error
int86	Invokes DOS interrupts
int86x	Invokes DOS interrupts with segment register values
intdos	Invokes DOS system call using registers other than DX and AL
intdosx	Invokes DOS system call using registers other than DX and AL with segment register values
segread	Returns current values of segment registers (OS/2 and DOS)

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 error information returned by DOS system call 0x59 (extended error handling). This function is provided for use with DOS versions 3.0 and later.

The **bdos** routine is useful for invoking 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 your 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 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 any interrupt. The **int86x** routine is similar; however, like the **intdosx** routine, it is designed to work with large-model programs and far items, as described in the preceding paragraph.

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. You can use these macros in OS/2 as well as DOS.

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_intr** 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 Encyclopedia* (Duncan, ed.; Redmond, WA: Microsoft Press, 1988)or the *Programmer's PC Sourcebook* (Hogan; Redmond, WA: Microsoft Press, 1988).

NOTE *The DOS interface I/O routines are generally incompatible with console, low-level, and stream I/O routines. Do not mix different types of I/O routines in the same source file.*

2.15 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.

Routine	Use
asctime	Converts time from type **struct tm** to a character string
clock	Returns the elapsed CPU time for a process
ctime	Converts time from a long integer to a character string

difftime	Computes the difference between two times
ftime	Puts current system time in variable of type **struct tm**
gmtime	Converts time from integer to **struct tm**
localtime	Converts time from integer to **struct tm** with local correction
mktime	Converts time to a calendar value
_strdate	Returns the current system date as a string
strftime	Formats a date and time string
_strtime	Returns the current system time as a string
time	Gets current system time as a long integer
tzset	Sets external time variables from the environment time variable
utime	Sets file-modification time

The **time** and **ftime** functions return the current time as the number of seconds elapsed since midnight Universal Coordinated Time (UTC) on January 1, 1970. This value can be converted, adjusted, and stored in a variety of ways by 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 files: SYS\TYPES.H and SYS\TIMEB.H. It is declared in SYS\TIMEB.H. The **utime** function also requires two include files: SYS\TYPES.H and SYS\UTIME.H. It 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. Section 3.2, which describes the global variables **daylight**, **timezone**, and **tzname**, includes a discussion of the TZ variable. TZ is also described on the **tzset** reference page in Part 2 of this book.

The **_strdate** and **_strtime** routines return strings containing the current date and time, respectively, in the DOS and OS/2 date and time format rather than in the XENIX-style formats.

The **stfrtime** function is useful for creating international versions of a program. See Section 2.8, "Internationalization."

2.16 Variable-Length Argument Lists

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 ANSI C standard.

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

For more information on the differences between the two versions and for an explanation of how to use the macros, see their descriptions in Part 2 of this book.

Global Variables and Standard Types

CHAPTER

3

The Microsoft C Run-Time Library contains definitions for a number of variables and standard 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.

3.1 _amblksiz

The **_amblksiz** variable controls memory heap granularity.

It is declared in the MALLOC.H include file as follows:

extern unsigned int _amblksiz;

The **_amblksiz** variable controls the amount of memory used in the heap for dynamic memory allocation.

Memory space is always requested from the operating system in blocks containing **_amblksiz** bytes. The first time a program calls a memory-allocation function such as **malloc**, the operating system allocates a block of heap memory. The size of this block is defined by **_amblksiz**, which has a default value of 8K (8,192 bytes).

Later memory requests are satisfied from the original block. When that block is exhausted, another block of **_amblksiz** bytes is allocated. If your C program allocates a block larger than **_amblksiz**, multiple blocks that are each of size **_amblksiz** are allocated until the request is satisfied.

To change the size of the default memory block, assign the desired size to the **_amblksiz** variable, as in the following example:

```
_amblksiz = 2048;
```

The heap allocator always rounds the operating-system request to the nearest power of 2 greater than or equal to **_amblksiz**. The above statement allocates memory in multiples of 2K (2,048 bytes).

Fewer system calls are required if you set **_amblksiz** to a large value, but your program may use more memory than needed. If program speed is important, set **_amblksiz** to a large value. If size is important, set **_amblksiz** to a smaller value.

Note that adjusting the value of **_amblksiz** affects allocation in the near, far, and based heaps. The value of **_amblksiz** has no effect on huge memory blocks (those allocated with **halloc** and similar functions).

3.2 daylight, timezone, tzname

The **daylight**, **timezone**, and **tzname** variables are global timezone variables used in time functions.

They are declared in the TIME.H include files as follows:

extern int daylight;

extern long timezone;

extern char *tzname [2];

Some time and date routines use the **daylight**, **timezone**, and **tzname** variables to make local-time adjustments. Whenever a program calls the **ftime**, **localtime**, or **tzset** function, the value of **daylight**, **timezone**, and **tzname** is determined from the value of the **TZ** environment variable. If you do not explicitly set the value of **TZ**, the default value of PST8PDT is used. The following list shows each variable and its value:

Variable	Value
daylight	Nonzero if a daylight-saving-time zone (DST) is specified in **TZ**; otherwise zero. Default value is one.
timezone	Difference in seconds between Greenwich mean time and the local time. Default value is 28,800.
tzname[0]	Three-letter time zone name derived from the **TZ** environment variable. Default value is "PST" (Pacific standard time).
tzname[1]	Three-letter daylight-saving-time zone name derived from the **TZ** environment variable. Default value is PDT. If the DST zone is omitted from **TZ**, **tzname[1]** is an empty string.

3.3 _doserrno, errno, sys_errlist, sys_nerr

The **_doserrno, errno, sys_errlist,** and **sys_nerr** variables contain error codes, and are used by the **perror** and **_strerror** routines to print error information.

These variables are declared in the STDLIB.H include file. Manifest constants for the **errno** variables are declared in the ERRNO.H include file. The declarations are as follows:

extern int _doserrno;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

The **errno** variable is set to an integer value to reflect the type of error that has occurred in a system-level call. Each **errno** value is associated with an error message, which can be printed with the **perror** routine or stored in a string with the **strerror** routine.

Note that only some routines set the **errno** variable. If a routine sets **errno,** the description of the routine in the reference section says so explicitly.

The value of **errno** reflects the error value for the last call that set **errno.** However, this value is not necessarily reset by later successful calls. To avoid confusion, test for errors 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 in DOS and OS/2. Some of the values in ERRNO.H are present to maintain compatibility with XENIX and UNIX operating systems.

The **errno** values in DOS and OS/2 are a subset of the values for **errno** in XENIX systems. Thus, the **errno** value is not necessarily the same as the actual error code returned by a DOS or OS/2 system call. To access the actual DOS and OS/2 error code, use the **_doserrno** variable, which contains this value.

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 DOS and OS/2 error-code equivalents. In other cases, the value of **_doserrno** is undefined.

The **syserrlist** variable is an array; the **perror** and **strerror** routines use it to process error information. The **sys_nerr** variable tells how many elements the **sys_errlist** array contains.

Table 3.1 gives the **errno** values for DOS and OS/2, the system error message for each value, and the value of each constant. Note that only the **ERANGE** and **EDOM** constants are specified in the ANSI standard.

Table 3.1 errno Values and Their Meanings

Constant	Meaning	Value
E2BIG	Argument list too long	7
EACCES	Permission denied	13
EBADF	Bad file number	9
EDEADLOCK	Resource deadlock would occur	36
EDOM	Math argument	33
EEXIST	File exists	17
EINVAL	Invalid argument	22
EMFILE	Too many open files	24
ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12
ENOSPC	No space left on device	28
ERANGE	Result too large	34
EXDEV	Cross-device link	18

3.4 _fmode

The **_fmode** variable controls the default file-translation mode.

It is declared in the STDLIB.H include file as follows:

extern int _fmode;

By default, the value of **_fmode** is **O_TEXT**, 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 the flag **O_BINARY** by linking with BINMODE.OBJ or by assigning it the **O_BINARY** value.

3.5 _osmajor, _osminor, _osmode, _osversion

The **_osmajor**, **_osminor**, **_osmode**, and **_osversion** variables specify the version number of the operating system or the current mode of operation.

They are declared in the STDLIB.H include file as follows:

extern unsigned char _osmajor;

extern unsigned char _osminor;

extern unsigned char _osmode;

extern unsigned char _osversion;

The **_osmajor**, **_osminor**, and **_osversion** variables specify the version number of DOS or OS/2 currently in use. The **_osmajor** variable holds the "major" version number and the **_osminor** variable stores the "minor" version number. Thus, under DOS version 3.20, **_osmajor** is 3 and **_osminor** is 20. The **_osversion** variable holds both values; its low byte contains the major version number and its high byte the minor version number.

These variables are useful for creating programs that run in different versions of DOS and OS/2. 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**.

The **_osmode** variable indicates whether the program is in OS/2 protected mode or in real mode (DOS or OS/2 real mode). An **_osmode** value of **DOS_MODE** indicates real mode operation and a value of **OS2_MODE** indicates protected operation.

3.6 environ

The **environ** variable is a pointer to the strings in the process environment.

It is declared in the STDLIB.H include file as follows:

extern char *environ [ ];

The **environ** variable provides access to memory areas containing process-specific information.

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 operating-system 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; its location in memory may also change, depending on the program's memory requirements. The **environ** variable is adjusted in these cases and always points to the correct table location.

3.7 _psp

The **_psp** variable contains the segment address of the program segment prefix (PSP) for the process.

It is declared in the STDLIB.H include file as follows:

extern unsigned int _psp;

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.

Note that the **_psp** variable is supported only in DOS.

3.8 Standard Types

A number of library routines use values whose types are defined in include files. In the following list, these types are described, and the include file defining each type is given.

Standard Type	Description
clock_t	The **clock_t** type, defined in TIME.H, stores time values. It is used by the **clock** function.
complex	The **complex** structure, defined in MATH.H, stores the real and imaginary parts of complex numbers. It 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 DOS system call 59H (available under DOS versions 3.0 and later).

exception

The **exception** structure, defined in MATH.H, stores error information for math routines. It 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. A buffer of this type is used by the **setjmp** and **longjmp** routines to save and restore the program environment.

lconv

The **lconv** type is a structure containing formatting rules for numeric values in different countries. It is defined in LOCALE.H.

onexit_t	The **onexit** routine is declared as an **onexit_t** pointer type, which is defined in STDLIB.H.
ptrdiff_t	The **ptrdiff_t** type is used for the signed integral result of the subtraction of two pointers.
REGS	The **REGS** union, defined in DOS.H, stores byte and word register values to be passed to and returned from calls to the DOS interface functions.
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.
size_t	The **size_t** type, defined in STDDEF.H and several other include files, is the unsigned integral result of the **sizeof** operator.
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 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.
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.

PART 2

Run-Time Functions

Run-Time Functions

The second part of this book is the reference section. It describes, in alphabetical order, each function of the run-time library provided with the Microsoft C Professional Development System.

Each reference entry gives syntax, return values, and other useful information about the library functions. Information on compatibility is supplied to assist you in writing portable programs.

About the Run-Time Reference

The following pages describe, in alphabetical order, the more than 400 functions in the Microsoft run-time library. In some cases, related routines are clustered in the same description. For example, the based, near, and far versions of **_heapwalk** are in the same discussion, as are the regular and long double versions of the math functions, such as **acos** and **atan**. Differences are noted where appropriate. Refer to Chapter 2, "Run-Time Routines by Category," or to the index to locate any function that does not appear in the expected position within the alphabetical reference.

The discussion of each function (or group of functions) is divided into the following sections:

- ***Description.*** Summarizes the routine's effect, names the include file(s) containing its declaration, illustrates the syntax, and briefly describes the arguments.
- ***Remarks.*** Gives a more detailed description of the routine and how it is used.
- ***Return Value.*** Describes the value returned by the routine.
- ***Compatibility.*** Tells whether the routine is compatible with ANSI C, MS-DOS, OS/2, UNIX, and XENIX.
- ***See Also.*** Names related routines.
- ***Example.*** Gives a complete program showing the use of the routine.

Description

Aborts the current process and returns an error code.

#include <process.h>	Required only for function declarations; use either
#include <stdlib.h>	PROCESS.H or STDLIB.H

void abort(void);

Remarks

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.

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

Return Value

The **abort** function does not return control to the caller. Rather, it terminates the process and, by default, returns an exit code of 3 to the parent process.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

In multithread libraries, the **abort** function does not call **raise(SIGABRT)**. Instead, it simply terminates the process with exit code 3.

See Also

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

Example

```
/* ABORT.C:  This tries to open a file and aborts if the attempt fails. */

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

```
void main()
{

   FILE *stream;

   if( (stream = fopen( "NOSUCHF.ILE", "r" )) == NULL )
   {
      perror( "Couldn't open file" );
      abort();
   }
   else
      fclose( stream );
}
```

Output

```
Couldn't open file: No such file or directory

abnormal program termination
```

Description Calculates the absolute value.

#include <stdlib.h> **#include <math.h>**	Required only for function declarations; use either STDLIB.H or MATH.H

int abs(int *n*);

n	Integer value

Remarks 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **cabs, fabs, labs**

Example

```
/* ABS.C: This program computes and displays the absolute values of
 * several numbers.
 */

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

void main()
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;

   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);

   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);

   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

Output

```
The absolute value of -4 is 4
The absolute value of -41567 is 41567
The absolute value of -3.141593 is 3.141593
```

Description

Determines file-access permission.

#include <io.h>	Required only for function declarations
#include <errno.h>	Required for definition of **errno** constants

int access(char **pathname*, **int** *mode* **);**

pathname	File or directory path name
mode	Permission setting

Remarks

With files, the **access** function determines whether 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 DOS and OS/2, 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **chmod, fstat, open, stat**

Example

```
/* ACCESS.C: This example uses access to check the file named "data"
 * to see if it exists and if writing is allowed.
 */

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

void main()
{
   /* Check for existence */
   if( (access( "access.c", 0 )) != -1 )
   {
      printf( "File exists\n" );

      /* Check for write permission */
      if( (access( "access.c", 2 )) != -1 )
         printf( "File has write permission\n" );
   }
}
```

Output

```
File exists
File has write permission
```

Description

Calculate the arccosine.

#include <math.h>

#include <errno.h> Required for definition of **errno** constant

double acos(double *x* **);**

long double acosl(long double *x* **);**

x Value whose arccosine is to be calculated

Remarks

The **acos** functions return the arccosine of *x* in the range 0 to π radians. The value of *x* must be between –1 and 1. The **acosl** function is the 80-bit counterpart, which uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

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

Compatibility

acos

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

acosl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

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

Example

```
/* ASINCOS.C: This program prompts for a value in the range -1 to 1.
 * Input values outside this range will produce DOMAIN error messages.
 * If a valid value is entered, the program prints the arcsine and the
 * arccosine of that value.
 */

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

```
void main()
{
   double x, y;

   printf( "Enter a real number between -1 and 1: " );
   scanf( "%lf", &x );
   y = asin( x );
   printf( "Arcsine of %f = %f\n", x, y );
   y = acos( x );
   printf( "Arccosine of %f = %f\n", x, y );
}
```

Output

```
Enter a real number between -1 and 1: .32696
Arcsine of 0.326960 = 0.333085
Arccosine of 0.326960 = 1.237711
```

Description

Allocates memory on the stack.

#include <malloc.h>	Required only for function declarations

void *alloca(size_t *size* **);**

size	Bytes to be allocated from stack

Remarks

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

When you compile with optimization on (either by default or by using one of the /O options), the stack pointer may not be restored properly in functions that have no local variables and that also reference the **alloca** function. The following program demonstrates the problem:

```
/* Compile with CL /Lp /AM /Ox /Fc */
#include <malloc.h>

void main( void )
{
   func( 10 );
}
void func( register int i )
{
   alloca( i );
}
```

To ensure that the stack pointer is properly restored, make sure that any function referencing **alloca** declares at least one local variable.

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.

Return Value

The **alloca** routine returns a **void** 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ☐ XENIX

See Also **calloc** functions, **malloc** functions, **realloc** functions

Example

```
/* ALLOCA.C: This program checks the stack space available before
 * and after using the alloca function to allocate space on the stack.
 */

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

void main()
{
   char *buffer;

   printf( "Bytes available on stack: %u\n", stackavail() );

   /* Allocate memory for string. */
   buffer = alloca( 120 * sizeof( char ) );
   printf( "Enter a string: " );
   gets( buffer );
   printf( "You entered: %s\n", buffer );

   printf( "Bytes available on stack: %u\n", stackavail() );
}
```

Output

```
Bytes available on stack: 2028
Enter a string: How much stack space will this string take?
You entered: How much stack space will this string take?
Bytes available on stack: 1902
```

Description Draw elliptical arcs.

#include <graph.h>

short _far _arc(short *x1*, **short** *y1*, **short** *x2*, **short** *y2*, **short** *x3*, **short** *y3*,
short *x4*, **short** *y4* **);**

short _far _arc_w(double *x1*, **double** *y1*, **double** *x2*, **double** *y2*, **double** *x3*, **double** *y3*,
double *x4*, **double** *y4* **);**

short _far _arc_wxy(struct _wxycoord _far **pwxy1*, **struct _wxycoord _far** **pwxy2*,
struct _wxycoord _far **pwxy3*, **struct _wxycoord _far** **pwxy4* **);**

x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
x3, *y3*	Second point of start vector (center of bounding rectangle is first point)
x4, *y4*	Second point of end vector (center of bounding rectangle is first point)
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
pwxy3	Second point of start vector (center of bounding rectangle is first point)
pwxy4	Second point of end vector (center of bounding rectangle is first point)

Remarks The **_arc** functions draw elliptical arcs. The center of the arc is the center of the bounding rectangle, which is defined by points (*x1*, *y1*) and (*x2*, *y2*) for **_arc** and **_arc_w** and by points *pwxy1* and *pwxy2* for **_arc_wxy**. The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for **_arc** and **_arc_w** and through *pwxy3* for **_arc_wxy**. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for **_arc** and **_arc_w** and through *pwxy4* for **_arc_wxy**.

The **_arc** routine uses the view coordinate system. The **_arc_w** and **_arc_wxy** functions use the real-valued window coordinate system.

In each case, the arc is drawn using the current color. Since an arc does not define a closed area, it is not filled.

Return Value These functions return a nonzero value if the arc is successfully drawn; otherwise, they return 0.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_ellipse** functions, **_lineto** functions, **_pie** functions, **_rectangle** functions, **_setcolor**

Example

```
/* ARC.C: This program draws a simple arc. */

#include <graph.h>
#include <stdlib.h>
#include <conio.h>

void main()
{
   short x, y;
   struct xycoord xystart, xyend, xyfill;

   /* Find a valid graphics mode */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   /* Draw arcs         */
   x = 100; y = 100;
   _arc( x - 60, y - 60, x, y, x - 30, y - 60, x - 60, y - 30 );
   _arc( x + 60, y + 60, x, y, x,      y + 30, x + 30, y );

   /* Get endpoints of second arc and enclose the figure, then fill it. */
   _getarcinfo( &xystart, &xyend, &xyfill );
   _moveto( xystart.xcoord, xystart.ycoord );
   _lineto( xyend.xcoord,   xyend.ycoord );
   _floodfill( xyfill.xcoord, xyfill.ycoord, _getcolor() );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description

Converts a **tm** time structure to a character string.

#include <time.h>

char *asctime(const struct tm ***_timeptr_ **);

timeptr	Time/date structure

Remarks

The **asctime** function converts a time stored as a structure to a character string. The *timeptr* 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 complete description of the **tm** structure fields.)

The **tm** structure contains the following elements:

Element	Description
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

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 newline character (**\n**) and the null character (**'\0'**) occupy the last two positions of the string. The **asctime** function uses a single statically allocated buffer to hold the return string. Each call to this routine destroys the result of the previous call.

Return Value

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

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

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

Example

```
/* ASCTIME.C: This program places the system time in the long integer aclock,
 * translates it into the structure newtime and then converts it to
 * string form for output, using the asctime function.
 */

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

struct tm *newtime;
time_t aclock;

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

   newtime = localtime( &aclock );     /* Convert time to struct tm form */

   /* Print local time as a string */
   printf( "The current date and time are: %s\n", asctime( newtime ) );
}
```

Output

```
The current date and time are: Thu Jun 15 06:57:59 1989
```

Description Calculate the arcsine.

#include <math.h>

#include <errno.h>

double asin(double *x* **);**

long double asinl(long double *x* **);**

x	Value whose arcsine is to be calculated

Remarks The **asin** functions calculate the arcsine of *x* in the range $-\pi/2$ to $\pi/2$ radians. The value of *x* must be between –1 and 1. The **asinl** function is the 80-bit counterpart, which uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value The **asin** functions return 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** (or **_matherrl**) routine.

Compatibility **asin**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

asinl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **acos** functions, **atan** functions, **cos** functions, **matherr**, **sin** functions, **tan** functions

Example

```
/* ASINCOS.C: This program prompts for a value in the range -1 to 1.
 * Input values outside this range will produce DOMAIN error messages.
 * If a valid value is entered, the program prints the arcsine and the
 * arccosine of that value.
 */

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

```
void main()
{
   double x, y;

   printf( "Enter a real number between -1 and 1: " );
   scanf( "%lf", &x );
   y = asin( x );
   printf( "Arcsine of %f = %f\n", x, y );
   y = acos( x );
   printf( "Arccosine of %f = %f\n", x, y );
}
```

Output

```
Enter a real number between -1 and 1: .32696
Arcsine of 0.326960 = 0.333085
Arccosine of 0.326960 = 1.237711
```

Description Prints an error message and aborts the program.

#include <assert.h>

#include <stdio.h>

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

expression	C expression specifying assertion being tested

Remarks 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).

The **assert** routine is typically used in program development 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 None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **abort, raise, signal**

Example

```
/* ASSERT.C: 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.
 */

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

```
void analyze_string( char *string );   /* Prototype */

void main()
{
   char  test1[] = "abc", *test2 = NULL, test3[] = "";

   printf ( "Analyzing string '%s'\n", test1 );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 );
   analyze_string( test2 );
   printf ( "Analyzing string '%s'\n", test3 );
   analyze_string( test3 );
}

/* Tests a string to see if it is NULL, empty, or longer than 0 characters */
void analyze_string( char * string )
{
   assert( string != NULL );        /* Cannot be NULL */
   assert( *string != '\0' );       /* Cannot be empty */
   assert( strlen( string ) > 2 );  /* Length must be greater than 2 */
}
```

Output

```
Analyzing string 'abc'
Analyzing string '(null)'
Assertion failed: string != NULL, file assert.c, line 28

abnormal program termination
```

Description Calculate the arctangent of x (**atan** and **atanl**) and the arctangent of y/x (**atan2** and **atan2l**).

#include <math.h>

double atan(double *x* **);**

double atan2(double *y*, **double** *x* **);**

long double atanl(long double *x* **);**

long double atan2l(long double *y*, **long double** *x* **);**

x, *y*	Any number

Remarks The **atan** family of functions calculates the arctangent of x, and the **atan2** family of functions calculates the arctangent of y/x. The **atan** group returns a value in the range $-\pi/2$ to $\pi/2$ radians, and the **atan2** group returns a value in the range $-\pi$ to π radians. The **atan2** functions use the signs of both arguments to determine the quadrant of the return value.

Return Value The **atan** family of functions returns the arctangent result. If both arguments of **atan2** or **atan2l** 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** (or **_matherrl**) routine.

Compatibility **atan, atan2**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

atanl, atan2l

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **acos** functions, **asin** functions, **cos** functions, **matherr**, **sin** functions, **tan** functions

Example

```
/* ATAN.C: This program calculates the arctangent of 1 and -1. */

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

```
void main()
{
   double x1, x2, y;

   printf( "Enter a real number: " );
   scanf( "%lf", &x1 );
   y = atan( x1 );
   printf( "Arctangent of %f: %f\n", x1, y );
   printf( "Enter a second real number: " );
   scanf( "%lf", &x2 );
   y = atan2( x1, x2 );
   printf( "Arctangent of %f / %f: %f\n", x1, x2, y );
}
```

Output

```
Enter a real number: -862.42
Arctangent of -862.420000: -1.569637
Enter a second real number: 78.5149
Arctangent of -862.420000 / 78.514900: -1.480006
```

Description Processes the specified function at exit.

#include <stdlib.h>	Required only for function declarations

int atexit(void (*func*)(void));

func	Function to be called

Remarks 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 in LIFO (last-in-first-out) order. No more than 32 functions can be registered with **atexit** or **onexit**. The functions passed to **atexit** cannot take parameters.

All routines passed to **atexit** should have the **_loadds** attribute if used in multithread dynamic-link libraries.

Return Value The **atexit** function returns 0 if it is successful, or a nonzero value if an error occurs (e.g., if there are already 32 exit functions defined).

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

Use the ANSI-standard **atexit** function (rather than the similar **onexit** function) whenever ANSI portability is desired.

In the OS/2 environment, the **atexit** function calls the OS/2 function **DosExitList**.

See Also **abort, exit, _exit, onexit**

Example

```
/* ATEXIT.C: 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.
 */

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

```
void fn1( void ), fn2( void ), fn3( void ), fn4( void );

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

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

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

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

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

Output

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

Description

Convert strings to double (**atof**), long double (**_atold**) integer (**atoi**), or long (**atol**).

#include <math.h>	**atof, _atold**
#include <stdlib.h>	**atof, _atold, atoi, atol**

double atof(const char ****string* **);

long double _atold(const char ****string* **);

int atoi(const char ****string* **);

long atol(const char ****string* **);

string	String to be converted

Remarks

These functions convert a character string to a double-precision floating-point value (**atof**), an integer value (**atoi**), a long integer value (**atol**), or a long double value (**_atold**). The input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

The string size that can be handled by the **atof** or **_atold** function is limited to 100 characters.

The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character (**'\0'**) terminating the string.

The **atof** and **_atold** functions expect *string* to have the following form:

[[*whitespace*]] [[{**sign**}]] [[IK0digits]] [[.*digits*]] [[{**d** | **D** | **e** | **E**}[[*sign*]]*digits*]]

A *whitespace* consists of space and/or tab characters, which are ignored; *sign* is either plus (+) or minus (–); 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, long double, int,** or **long** value produced by interpreting the input characters as a number. The return value is 0 (for **atoi**), 0L (for **atol**), and 0.0 (for **atof** and **_atold**) if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

Compatibility

atof, atoi, atol

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_atold

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

ecvt, fcvt, gcvt

Example

```
/* ATOF.C: This program shows how numbers stored as strings can be
 * converted to numeric values using the atof, atoi, and atol functions.
 */

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

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

   s = "  -2309.12E-15";    /* Test of atof */
   x = atof( s );
   printf( "atof test:  ASCII string: %s\tfloat:     %e\n", s, x );

   s = "7.8912654773d210";  /* Test of atof */
   x = atof( s );
   printf( "atof test:  ASCII string: %s\tfloat:     %e\n", s, x );

   s = "  -9885 pigs";      /* Test of atoi */
   i = atoi( s );
   printf( "atoi test:  ASCII string: %s\t\tinteger: %d\n", s, i );

   s = "98854 dollars";     /* Test of atol */
   l = atol( s );
   printf( "atol test:  ASCII string: %s\t\tlong:    %ld\n", s, l );
}
```

Output

```
atof test:  ASCII string:   -2309.12E-15        float:     -2.309120e-012
atof test:  ASCII string: 7.8912654773d210      float:      7.891265e+210
atoi test:  ASCII string:   -9885 pigs          integer: -9885
atol test:  ASCII string: 98854 dollars         long:    98854
```

Description

Invokes the DOS system call.

#include <dos.h>

int bdos(int *dosfunc*, **unsigned int** *dosdx*, **unsigned int** *dosal* **);**

dosfunc	Function number
dosdx	DX register value
dosal	AL register value

Remarks

The **bdos** function invokes the DOS system call specified by *dosfunc* 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 is complete, **bdos** returns the contents of the AX register.

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

Do not use the **bdos** function to call interrupts that modify the DS register. Instead, use the **intdosx** or **int86x** function. The **intdosx** and **int86x** functions load the DS and ES registers from the *segregs* parameter and also store the DS and ES registers into *segregs* after the function call.

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, your program cannot determine whether the return value is an error code. The **intdos** function should be used in these cases.

Return Value

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

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

intdos, intdosx

Example

```
/* BDOS.C: This example calls DOS function 0x9 (display string)
 * to display a $-terminated string.
 */

#include <dos.h>
```

```
/* Function 0x09 assumes that DS will contain segment of the string.
 * This will be true for all memory models if the string is declared near.
 */
char _near str[] = "Hello world!\r\n$";

void main()
{
   /* Offset of string must be in DX, segment in DS. AL is not needed,
    * so 0 is used.
    */
   bdos( 0x09, (int)str, 0 );
}
```

Output

```
Hello world!
```

Description

Begins thread in OS/2 process.

#include <process.h>	Multithread version of PROCESS.H
#include <stddef.h>	Declaration of *threadid* variable

int _far _beginthread(void(_far ****start_address* **)(void _far *),
void _far ****stack_bottom*, **unsigned *stack_size*, **void _far ****arglist* **);**

start_address	Starting address
stack_bottom	Address of the thread stack
stack_size	Stack size for thread
arglist	Argument list for thread

Remarks

The **_beginthread** function creates a thread that begins execution of a far routine at *start_address*. When the thread returns from that far routine, it is terminated automatically. The user can also terminate the thread by calling **_endthread**.

The address of the thread stack is given by *stack_bottom*. If *stack_bottom* is set to **NULL**, the run-time library code will allocate and deallocate the thread stack as needed. Since the **_beginthread** function can determine the current status of all thread IDs, it can free the old stack and allocate a new stack whenever a thread is reused.

If it is not **NULL**, the *stack_bottom* argument must specify a word address, and the stack must be at least as long as specified by the *stack_size* argument. Usually this memory is either a global array or memory returned by **malloc** or **_fmalloc**.

The *stack_size* argument must be even and nonzero.

If you are writing multithread programs that make C run-time calls from child threads, be sure to allocate a sufficiently large stack. For example, the C function **printf** requires more than 500 bytes of stack space. To be safe, allocate at least 2,048 bytes for a thread's stack. (If your child thread makes no run-time calls, stack space is generally not a problem.)

As a general rule, you should have 2K of stack space free when calling any API (Applications Program Interface) routine (e.g., OS/2 system calls).

The *arglist* is a parameter, the size of a far pointer, to be passed to the newly created thread. Typically it is the address of a data item, such as a character string, to be passed to the new thread. The *arglist* may be **NULL** if not needed, but **_beginthread** should be provided with some value to pass to the child thread.

All threads will be terminated if any thread calls **abort**, **exit**, **_exit**, or **DosExit**. A good practice in multithread programming is to make the first thread the main thread and wait until other threads have terminated before exiting the program.

The OS/2 function **DosCreateThread** should not be called directly to create threads. The **_beginthread** function performs initialization procedures required to call other C run-time library functions safely.

Return Value

The function returns the thread identification number of the new thread, if successful. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EAGAIN	Too many threads
EINVAL	Invalid argument, "bad stack"

Compatibility

☐ ANSI ☐ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_endthread

Example

```
/* BEGTHRD.C illustrates multiple threads using functions:
 *      _beginthread            _endthread
 *
 * Also the global variable:
 *      _threadid
 *
 * This program requires the multithread library. For example, compile
 * with the following command line:
 *      CL /MT THREADS.C
 */

#define INCL_NOCOMMON
#define INCL_NOPM
#define INCL_DOSPROCESS
#define INCL_VIO
#include <os2.h>
#include <process.h>    /* _beginthread, _endthread */
#include <stddef.h>     /* _threadid                */
#include <stdlib.h>
#include <conio.h>

void Bounce( int c );       /* Prototypes */
void CheckKey( void *dummy );
```

```
/* GetRandom returns a random integer between min and max. */
#define GetRandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))

#define STACK_SIZE   4096

BOOL repeat = TRUE;          /* Global repeat flag and video variable */
VIOMODEINFO vmi = { sizeof( VIOMODEINFO ) };

void main()
{
    PCHAR   stack;
    CHAR    ch = 'A';

    /* Get display screen's text row and column information. */
    VioGetMode( &vmi, 0 );

    /* Launch CheckKey thread to check for terminating keystroke. */
    _beginthread( CheckKey, NULL, STACK_SIZE, NULL );

    /* Loop until CheckKey terminates program. */
    while( repeat )
    {
        /* On first loops, launch character threads. */
        _beginthread( Bounce, NULL, STACK_SIZE, (void *)ch++ );

        /* Wait one second between loops. */
        DosSleep( 1000L );
    }
}

/* CheckKey - Thread to wait for a keystroke, then clear repeat flag. */
void CheckKey( void *dummy )
{
    getch();
    repeat = 0;      /* _endthread implied */
}

/* Bounce - Thread to create and control a colored letter that moves
 * around on the screen.
 *
 * Params: ch - the letter to be moved
 */
void Bounce( int ch )
{
    /* Generate letter and color attribute from thread argument. */
    char      blankcell[2] = { 0x20, 0x07 };
    char      blockcell[2] = { ch , (ch % 16) + 1 };
    int       xold, xcur, yold, ycur;
    BOOL      first = TRUE;
```

```
    /* Seed random number generator and get initial location. */
    srand( *_threadid );
    xcur = GetRandom( 0, vmi.col - 1 );
    ycur = GetRandom( 0, vmi.row - 1 );
    while( repeat )
    {
        /* Pause between loops. */
        DosSleep( 100L );

        /* Blank out our old position on the screen, and draw new letter. */
        if( first )
            first = FALSE;
        else
            VioWrtCellStr( blankcell, 2, yold, xold, 0 );
        VioWrtCellStr( blockcell, 2, ycur, xcur, 0 );

        /* Increment the coordinate for next placement of the block. */
        xold = xcur;
        yold = ycur;
        xcur += GetRandom( -1, 1 );
        ycur += GetRandom( -1, 1 );

        /* Correct placement (and beep) if about to go off the screen. */
        if( xcur < 0 )
            xcur = 1;
        else if( xcur == vmi.col )
            xcur = vmi.col - 2;
        else if( ycur < 0 )
            ycur = 1;
        else if( ycur == vmi.row )
            ycur = vmi.row - 2;

        /* If not at screen border, continue, otherwise beep. */
        else
            continue;
        DosBeep( (ch - 'A') * 100, 175 );
    }
    /* _endthread given (but not really needed) to terminate. */
    _endthread();
}
```

Description

Compute the Bessel function.

```
#include <math.h>

double j0( double x );
double j1( double x );
double jn( int n, double x );
double y0( double x );
double y1( double x );
double yn( int n, double x );
long double _j0l( long double x );
long double _jnl( int n, long double x );
long double _j1l( long double x );
long double _y0l( long double x );
long double _y1l( long double x );
long double _ynl( int n, long double x );
```

x	Floating-point value
n	Integer order

Remarks

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.

The long double versions of these functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

The Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun; Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

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** (or **_matherrl**) routine.

Compatibility

j0, j1, jn, y0, y1, yn

☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_j0l, _j1l, _jnl, _y0l, _y1l, _ynl

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

matherr

Example

```
/* BESSEL.C: This program illustrates Bessel functions, including:
 *      j0          j1          jn          y0          y1          yn
 */

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

void main()
{
    double x = 2.387;
    int n = 3, c;

    printf( "Bessel functions for x = %f:\n", x );
    printf( "  Kind\t\tOrder\t\Function\tResult\n\n" );
    printf( "  First\t\t0\tj0( x )\t\t%f\n", j0( x ) );
    printf( "  First\t\t1\tj1( x )\t\t%f\n", j1( x ) );
    for( c = 2; c < 5; c++ )
        printf( "  First\t\t%d\tjn( n, x )\t%f\n", c, jn( c, x ) );

    printf( "  Second\t0\ty0( x )\t\t%f\n", y0( x ) );
    printf( "  Second\t1\ty1( x )\t\t%f\n", y1( x ) );
    for( c = 2; c < 5; c++ )
        printf( "  Second\t%d\tyn( n, x )\t%f\n", c, yn( c, x ) );
}
```

Output

```
Bessel functions for x = 2.387000:
  Kind          Order   Function        Result

  First         0       j0( x )         0.009288
  First         1       j1( x )         0.522941
  First         2       jn( n, x )      0.428870
  First         3       jn( n, x )      0.195734
  First         4       jn( n, x )      0.063131
  Second        0       y0( x )         0.511681
  Second        1       y1( x )         0.094374
  Second        2       yn( n, x )      -0.432608
  Second        3       yn( n, x )      -0.819314
  Second        4       yn( n, x )      -1.626833
```

Description

Frees a specified based heap.

#include <malloc.h>	Required only for function declarations

int _bfreeseg(_segment *seg* **);**

seg	Segment selected

Remarks

The **_bfreeseg** function frees a based heap. The *seg* argument is a based heap returned by an earlier call to **_bheapseg**. It specifies the based heap to be freed.

The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed heap is again available for allocation.

Return Value

The **_bfreeseg** function returns 0 if successful and –1 in the case of an error.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_bheapseg, calloc functions, **free** functions, **malloc** functions, **realloc** functions

Example

```
/* BHEAPSEG.C: This program C illustrates dynamic allocation of based
 * memory using functions _bheapseg, _bfreeseg, _bmalloc, and _bfree.
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

void main()
{
    _segment seg;
    char _based( seg ) *outstr, _based( seg ) *instr;
    char _based( seg ) *pout,   _based( seg ) *pin;
    char tmpstr[80];
    int  len;

    printf( "Enter a string: " );
    gets( tmpstr );

    /* Request a based heap. Use based so that memory won't be taken from
     * near heap.
     */
```

```
    if( (seg = _bheapseg( 1000 )) == _NULLSEG )
        exit( 1 );

    /* Allocate based memory for two strings. */
    len = strlen( tmpstr );
    if( ((instr  = _bmalloc( seg, len + 1 )) == _NULLOFF) ||
        ((outstr = _bmalloc( seg, len + 1 )) == _NULLOFF) )
        exit( 1 );

    /* Copy a lowercased string to dynamic memory. The based memory is
     * far when addressed as a whole.
     */
    _fstrlwr( _fstrcpy( (char _far *)instr, (char _far *)tmpstr ) );

    /* Copy input string to output string in reversed order. When reading
     * and writing individual characters from a based heap, the compiler will
     * try to process them as near, thus speeding up the processing.
     */
    for( pin = instr + len - 1, pout = outstr;
                pout < outstr + len; pin--, pout++ )
        *pout = *pin;
    *pout = '\0';

    /* Display strings. Again strings as a whole are far. */
    printf( "Input:  %Fs\n", (char _far *)instr );
    printf( "Output: %Fs\n", (char _far *)outstr );

    /* Free blocks and release based heap. */
    _bfree( seg, instr );
    _bfree( seg, outstr );
    _bfreeseg( seg );
}
```

Output

```
Enter a string: Was I god
Input:  was i god
Output: dog i saw
```

Description

Allocates a based heap.

#include <malloc.h> Required only for function declarations

_segment _bheapseg(size_t *size* **);**

size Segment size to allocate

Remarks

The **_bheapseg** function allocates a based-heap segment of at least *size* bytes. (The block may be larger than *size* bytes because of space required for alignment and for maintenance information.)

The heap code will try to enlarge the heap as necessary. If the original block of memory is depleted (e.g., by calls to **_bmalloc** and **_brealloc**), the run-time code will try to enlarge the heap as necessary.

The value returned by **_bheapseg** is the identifier of the based-heap segment. This value should be saved and used in subsequent calls to other based-heap functions.

The **_bheapseg** function can be called repeatedly. For each call, the C library will allocate a new based-heap segment.

Return Value

The **_bheapseg** function returns the newly allocated segment selector that the user must save for use in subsequent based-heap functions. A return value of –1 indicates failure.

Always check the return from the **_bheapseg** function (especially when it is used in real mode), even if the amount of memory requested is small.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

calloc functions, **free** functions, **malloc** functions, **realloc** functions

Example

```
/* BHEAPSEG.C: This program C illustrates dynamic allocation of based
 * memory using functions _bheapseg, _bfreeseg, _bmalloc, and _bfree.
 */

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

```
void main()
{
    _segment seg;
    char _based( seg ) *outstr, _based( seg ) *instr;
    char _based( seg ) *pout,   _based( seg ) *pin;
    char tmpstr[80];
    int  len;

    printf( "Enter a string: " );
    gets( tmpstr );

    /* Request a based heap. Use based so that memory won't be taken from
     * near heap.
     */
    if( (seg = _bheapseg( 1000 )) == _NULLSEG )
        exit( 1 );

    /* Allocate based memory for two strings. */
    len = strlen( tmpstr );
    if( ((instr  = _bmalloc( seg, len + 1 )) == _NULLOFF) ||
        ((outstr = _bmalloc( seg, len + 1 )) == _NULLOFF) )
        exit( 1 );

    /* Copy a lowercased string to dynamic memory. The based memory is
     * far when addressed as a whole.
     */
    _fstrlwr( _fstrcpy( (char _far *)instr, (char _far *)tmpstr ) );

    /* Copy input string to output string in reversed order. When reading
     * and writing individual characters from a based heap, the compiler will
     * try to process them as near, thus speeding up the processing.
     */
    for( pin = instr + len - 1, pout = outstr;
                pout < outstr + len; pin--, pout++ )
        *pout = *pin;
    *pout = '\0';

    /* Display strings. Again, strings as a whole are far. */
    printf( "Input:  %Fs\n", (char _far *)instr );
    printf( "Output: %Fs\n", (char _far *)outstr );

    /* Free blocks and release based heap. */
    _bfree( seg, instr );
    _bfree( seg, outstr );
    _bfreeseg( seg );
}
```

Output

```
Enter a string: Was I god
Input:  was i god
Output: dog i saw
```

Description

Calls BIOS disk services using system call INT 0x13.

#include <bios.h>

unsigned _bios_disk(unsigned *service*, **struct diskinfo_t** **diskinfo* **);**

service	Disk function desired
diskinfo	Disk parameters

Remarks

The **_bios_disk** routine uses system call INT 0x13 to provide several disk-access functions. The *service* parameter selects the function desired, while the *diskinfo* structure provides the necessary parameters. Note that the low-level disk operations allowed by the **_bios_disk** routine are very dangerous to use because they allow direct manipulation of the disk.

The *diskinfo* structure provides the following parameters:

Element	Description
unsigned drive	Drive number
unsigned head	Head number
unsigned track	Track number
unsigned sector	Starting sector number
unsigned nsectors	Number of sectors to read, write, or compare
void far *buffer	Memory location to write to, read from, or compare

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

Constant	Function
_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 a technical reference to the PC BIOS to determine the marker format. There is no return value.

_DISK_READ

Reads one or more disk sectors into memory. This service uses all fields of the structure pointed to by *diskinfo*, as defined earlier in this section. 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. If there is an error, the high-order byte will contain a set of status flags, as defined under **_DISK_READ**. Status is returned in the 8 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**	Direct Memory Access (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

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

_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 pointed to by *diskinfo*, as defined earlier in this section. 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_READ** (above).

_DISK_WRITE Writes data from memory to one or more disk sectors. This service uses all fields of the structure pointed to by *diskinfo*, as defined earlier in this section. 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_READ** (above).

Return Value The **_bios_disk** function returns the value in the AX register after the BIOS interrupt.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

Example

```
/* BDISK.C: 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.
 */

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

void 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;
```

```
    printf( "Insert disk in drive A: and press any key\n" );
    getch();
    status = _bios_disk( _DISK_VERIFY, &disk_info );
    printf( "Return value: 0x%.4x\n", status );
    if( status & 0xff00 )       /* Error if high byte is 0 */
       printf( "Seek error\n" );
    else
       printf( "No seek error\n" );

    printf( "Press any key\n" );
    getch();
    disk_info.head = 0;         /* Valid head number */
    status = _bios_disk( _DISK_VERIFY, &disk_info );
    printf( "Return value: 0x%.4x\n", status );
    if( status & 0xff00 )       /* Error if high byte is 0 */
       printf( "Seek error\n" );
    else
       printf( "No seek error\n" );
}
```

Output

```
Insert disk in drive A: and press any key
Return value: 0x0400
Seek error
Press any key
Return value: 0x0008
No seek error
```

Description

Calls BIOS equipment-list service, using system call INT 0x11.

#include <bios.h>

unsigned _bios_equiplist(void);

Remarks

The **_bios_equiplist** routine uses system call 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
1	True (1) if math coprocessor installed
2–3	System RAM in 16K blocks (16–64K)
4–5	Initial video mode
6–7	Number of floppy-disk drives installed (00 = 1, 01 = 2, etc.)
8	False (0) if and only if a Direct Memory Access (DMA) chip is installed
9–11	Number of RS232 serial ports installed
12	True (1) if and only if a game adapter is installed
13	True (1) if and only if an internal modem is installed
14–15	Number of printers installed

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* BEQUIPLI.C: This program checks for the presence of diskettes. */

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

```
void main()
{
   unsigned equipment;

   equipment = _bios_equiplist();
   printf( "Equipment bits: 0x%.4x\n", equipment );
   if( equipment & 0x1000 )      /* Check for game adapter bit */
      printf( "Game adapter installed\n" );
   else
      printf( "No game adapter installed\n" );
}
```

Output

```
Equipment bits: 0x4061
No game adapter installed
```

Description

Calls BIOS keyboard services, using INT 0x16.

#include <bios.h>

unsigned _bios_keybrd(unsigned *service* **);**

service	Keyboard function desired

Remarks

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

Constant	Meaning
_KEYBRD_READ, _NKEYBRD_READ	Reads the next character 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, the call contains the ASCII value of the character typed. The high-order byte contains the keyboard scan code for the character. The **_NKEYBRD_READ** constant is used with enhanced keyboards to obtain the scan codes for function keys F11 and F12 and the cursor control keys.
_KEYBRD_READY, _NKEYBRD_READY	Checks whether a keystroke is waiting to be read and, if so, reads it. The return value is 0 if no keystroke is waiting, or it is the character waiting to be read, in the same format as the **_KEYBRD_READ** or **_NKEYBRD_READY** return. This service does not remove the waiting character from the input buffer, as does the **_KEYBRD_READ** or **_NKEYBRD_READ** service. The **_NKEYBRD_READY** constant is used with enhanced keyboards to obtain the scan codes for function keys F11 and F12 and the cursor control keys.

_KEYBRD_SHIFTSTATUS, _NKEYBRD_SHIFTSTATUS	Returns the current SHIFT-key status. Only the low-order byte of the return value is affected. The **_NKEYBRD_SHIFTSTATUS** constant is used to get a full 16-bit status value. Any combination of the following bits may be set:

Bit	Meaning if True
00H	Rightmost SHIFT key pressed
01H	Leftmost SHIFT key pressed
02H	Either CTRL key pressed
3H	Either ALT key pressed
04H	SCROLL LOCK on
05H	NUM LOCK on
06H	CAPS LOCK on
07H	In insert mode (INS)
08H	Left CTRL key pressed
09H	Left ALT key pressed
0AH	Right CTRL key pressed
0BH	Right ALT key pressed
0CH	SCROLL LOCK key pressed
0DH	NUM LOCK key pressed
0EH	CAPS LOCK key pressed
0FH	SYS REQ key pressed

Return Value

With the **...READ** and **...SHIFTSTATUS** arguments, the **_bios_keybrd** function returns the contents of the AX register after the BIOS call.

With the **...READY** argument, **_bios_keybrd** returns 0 if there is no key. If there is a key, **_bios_keybrd** returns the key waiting to be read (i.e. the same value as **_KEYBRD_READ**).

With the **...READ** and the **...READY** arguments, the **_bios_keybrd** function returns –1 if CTRL+BREAK has been pressed and is the next keystroke to be read.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* BKEYBRD.C: This program prints a message on the screen until the
 * right SHIFT key is pressed.
 */

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

void main()
{
   while( !(_bios_keybrd( _KEYBRD_SHIFTSTATUS ) & 0001) )
      printf( "Use the right SHIFT key to stop this message\n" );
   printf( "Right SHIFT key pressed\n" );
}
```

Output

```
Use the right SHIFT key to stop this message
Use the right SHIFT key to stop this message
Use the right SHIFT key to stop this message
Use the right SHIFT key to stop this message
Right SHIFT key pressed
```

Description Calls the BIOS memory-size service, using system call INT 0x12.

#include <bios.h>

unsigned _bios_memsize(void);

Remarks The **_bios_memsize** routine uses system call INT 0x12 to determine the total amount of main memory installed.

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.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

Example

```
/* BMEMSIZE.C: This program displays the amount of memory installed. */

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

void main()
{
   unsigned memory;

   memory = _bios_memsize();
   printf ( "The amount of memory installed is: %dK\n", memory );
}
```

Output

```
The amount of memory installed is: 639K
```

Description

Calls BIOS printer services using system call INT 0x17.

#include <bios.h>

unsigned _bios_printer(unsigned *service*, **unsigned** *printer*, **unsigned** *data* **);**

service	Printer function desired
printer	Target printer port
data	Output data

Remarks

The **_bios_printer** routine uses system call INT 0x17 to perform printer output services for parallel printers. The *printer* argument specifies the affected printer, where 0 is LPT1, 1 is LPT2, and so forth.

Some printers do not support the full set of signals. As a result, the "Out of Paper" condition, for example, may not be returned to your program.

The *service* argument can be any of the following manifest constants:

Constant	Meaning
_PRINTER_INIT	Initializes the selected printer. The *data* argument is ignored. The return value is the low-order status byte defined below.
_PRINTER_STATUS	Returns the printer status in the low-order status byte defined below. The *data* argument is ignored.
_PRINTER_WRITE	Sends the low-order byte of *data* to the printer specified by *printer*. 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

Return Value The **_bios_printer** function returns the value in the AX register after the BIOS interrupt.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* BPRINTER.C: This program checks the status of the printer attached to
 * LPT1 when it is off line, then initializes the printer.
 */

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

#define LPT1 0

void main()
{
   unsigned status;

   printf ( "Place printer off line and press any key\n" );
   getch();

   status = _bios_printer( _PRINTER_STATUS, LPT1, 0 );
   printf( "Status with printer off line: 0x%.4x\n\n", status );
   printf( "Put the printer on line and then\n" );
   printf( "Press any key to initialize printer\n" );
   getch();

   status = _bios_printer( _PRINTER_INIT, LPT1, 0 );
   printf( "Status after printer initialized: 0x%.4x\n", status );
}
```

Output

```
Place printer off line and press any key
Status with printer off line: 0x0018

Put the printer on line and then
Press any key to initialize printer
Status after printer initialized: 0x0090
```

Description

Calls BIOS communications services, using system call INT 0x14.

#include <bios.h>

unsigned _bios_serialcom(unsigned *service*, **unsigned** *serial_port*, **unsigned** *data* **);**

service	Communications service
serial_port	Serial port to use
data	Port configuration bits

Remarks

The **_bios_serialcom** routine uses system call 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 **_bios_serialcom** routine may not be able to establish reliable communications at baud rates in excess of 1,200 baud (**_COM_1200**) due to the overhead associated with servicing computer interrupts. Faster data communication rates are possible with more direct programming of serial-port controllers. See *C Programmer's Guide to Serial Communications* for more details on serial-communications programming in C.

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 combining (with the OR operator) 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	1,200 baud
_COM_2400	2,400 baud
_COM_4800	4,800 baud
_COM_9600	9,600 baud

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

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 have the following meanings:

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, any of the bits 9, 10, 11, or 15 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

Note that this function works only with IBM personal computers and true compatibles.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* BSERIALC.C: This program checks the status of serial port COM1. */

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

void main()
{
   unsigned com1_status;

   com1_status = _bios_serialcom( _COM_STATUS, 0, 0 );
   printf ( "COM1 status: 0x%.4x\n", com1_status );
}
```

Output

```
COM1 status: 0x6000
```

Description Calls BIOS time and date services, using system call INT 0x1A.

#include <bios.h>

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

service	Time function desired
timeval	Clock count

Remarks The **_bios_timeofday** routine uses system call INT 0x1A to get or set the 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 pointed to by *timeval*. If midnight has not passed since the last time the system clock was read or set, the function returns 0; otherwise, the function returns 1.
_TIME_SETCLOCK	Sets the current value of the system clock to the value in the location pointed to by *timeval*. There is no return value.

Return Value The **_bios_timeofday** function returns the value in the AX register after the BIOS interrupt.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* BTIMEOFD.C: This program gets the current system clock count before and after
 * a "do-nothing" loop and displays the difference.
 */

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

```
void 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 <= 900000; i++ )
      ;
   _bios_timeofday( _TIME_GETCLOCK, &end_tick );
   printf( "Ending tick count:    %lu\n", end_tick );
   printf( "Elapsed ticks:        %lu\n", end_tick - begin_tick );
}
```

Output

```
Beginning tick count: 1114255
Ending tick count:    1114287
Elapsed ticks:        32
```

Description

Performs binary search of a sorted array.

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

void *bsearch(const void *_key_, const void *_base_, size_t _num_, size_t _width_,
int (*_compare_)(const void *_elem1_, const void *_elem2_));

key	Object to search for
base	Pointer to base of search data
num	Number of elements
width	Width of elements
compare	Function that compares two elements: *elem1* and *elem2*
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

Remarks

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 calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The routine compares the elements, then returns one of the following values:

Value	Meaning
< 0	*elem1* less than *elem2*
= 0	*elem1* identical to *elem2*
> 0	*elem1* greater than *elem2*

If the array you are searching is not in ascending sort order, **bsearch** does not work properly. If the array contains duplicate records with identical keys, there is no way to predict which of the duplicate records will be located by **bsearch**.

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**.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **lfind, lsearch, qsort**

Example

```
/* BSEARCH.C: This program reads the command-line arguments, sorting them
 * with qsort, and then uses bsearch to find the word "cat."
 */

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

int compare( char **arg1, char **arg2 );  /* Declare a function for compare */

void main( int argc, char **argv )
{

   char **result;
   char *key = "cat";
   int i;

   /* Sort using Quicksort algorithm: */
   qsort( (char *)argv, argc, sizeof( char * ), compare );

   for( i = 0; i < argc; ++i )        /* Output sorted list */
      printf( "%s ", argv[i] );

   /*  Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch( (char *) &key, (char *)argv, argc,
                              sizeof( char * ), compare );
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}

int compare( char **arg1, char **arg2 )
{
   /* Compare all of both strings: */
   return strcmpi( *arg1, *arg2 );
}
```

Output

```
[C:\LIBREF] bsearch dog pig horse cat human rat cow goat
bsearch cat cow dog goat horse human pig rat
cat found at 0292:0FD0
```

Description Calculate absolute value of a complex number.

#include <math.h>

double cabs(struct complex *z* **);**

long double cabsl(struct _complexl *z* **);**

z	Complex number

Remarks The **cabs** and **cabsl** functions calculate the absolute value of a complex number, which must be a structure of type **complex** (or **_complexl**). The structure *z* is composed of a real component *x* and an imaginary component *y*. A call to one of the **cabs** routines is equivalent to the following:

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

The **cabsl** function is the 80-bit counterpart and it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value On overflow, these functions call **matherr** or **_matherrl**, return **HUGE_VAL**, and set **errno** to **ERANGE**.

Compatibility **cabs**

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

cabsl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **abs, fabs, labs**

Example

```
/* CABS.C: Using cabs, this program calculates the absolute value of
 * a complex number.
 */

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

```
void main()
{
   struct complex number = { 3.0, 4.0 };
   double d;

   d = cabs( number );
   printf( "The absolute value of %f + %fi is %f\n",
           number.x, number.y, d );
}
```

Output

```
The absolute value of 3.000000 + 4.000000i is 5.000000
```

Description

Allocate an array in memory with elements initialized to 0.

#include <stdlib.h>	For ANSI compatibility (**calloc** only)
#include <malloc.h>	Required only for function declarations

void *calloc(size_t *num*, **size_t** *size* **);**

void _based(void) *_bcalloc(_segment *seg*, **size_t** *num*, **size_t** *size* **);**

void _far *_fcalloc(size_t *num*, **size_t** *size* **);**

void _near *_ncalloc(size_t *num*, **size_t** *size* **);**

num	Number of elements
size	Length in bytes of each element
seg	Segment selector

Remarks

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

In large data models (compact-, large-, and huge-model programs), **calloc** maps to **_fcalloc**. In small data models (tiny-, small-, and medium-model programs), **calloc** maps to **_ncalloc**.

The various **calloc** functions allocate storage space in the data segments shown in the list below:

Function	Data Segment
calloc	Depends on data model of program
_bcalloc	Based heap, specified by *seg* segment selector
_fcalloc	Far heap (outside default data segment)
_ncalloc	Near heap (inside default data segment)

Return Value

The **calloc** functions return 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 **_fcalloc** and **_ncalloc** functions return **NULL** if there is insufficient memory available or if *num* or *size* is 0. The **_bcalloc** function returns **_NULLOFF** in this case.

Compatibility

calloc

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_bcalloc, _fcalloc, _ncalloc

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **free** functions, **halloc**, **hfree**, **malloc** functions, **realloc** functions

Example

```
/* CALLOC.C: This program uses calloc to allocate space for 40 long integers.
 * It initializes each element to zero.
 */

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

void main()
{
   long *buffer;

   buffer = (long *)calloc( 40, sizeof( long ) );
   if( buffer != NULL )
      printf( "Allocated 40 long integers\n" );
   else
      printf( "Can't allocate memory\n" );
   free( buffer );
}
```

Output

```
Allocated 40 long integers
```

Description Calculate the ceiling of a value.

#include <math.h>

double ceil(double *x* **);**

long double ceill(long double *x* **);**

x	Floating-point value

Remarks The **ceil** and **ceill** functions return a **double** (or **long double**) value representing the smallest integer that is greater than or equal to *x*.

The **ceill** function is the 80-bit counterpart and it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value These functions return the **double** or **long double** result. There is no error return.

Compatibility **ceil**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

ceill

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **floor, fmod**

Example

```
/* FLOOR.C: This example displays the largest integers less than or equal
 * to the floating-point values 2.8 and -2.8. It then shows the smallest
 * integers greater than or equal to 2.8 and -2.8.
 */

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

```
void main()
{
   double y;

   y = floor( 2.8 );
   printf( "The floor of 2.8 is %f\n", y );
   y = floor( -2.8 );
   printf( "The floor of -2.8 is %f\n", y );

   y = ceil( 2.8 );
   printf( "The ceil of 2.8 is %f\n", y );
   y = ceil( -2.8 );
   printf( "The ceil of -2.8 is %f\n", y );
}
```

Output

```
The floor of 2.8 is 2.000000
The floor of -2.8 is -3.000000
The ceil of 2.8 is 3.000000
The ceil of -2.8 is -2.000000
```

Description

Perform clean-up operations and return without terminating the process.

#include <process.h>

void _cexit(void);

void _c_exit(void);

Remarks

The **_cexit** function calls, in LIFO ("last in, first out") order, the functions registered by **atexit** and **onexit**. Then the **_cexit** function flushes all I/O buffers and closes all open files before returning.

The **_c_exit** function returns to the calling process without processing **atexit** or **onexit** functions or flushing stream buffers.

The behavior of the **exit**, **_exit**, **_cexit**, and **_c_exit** functions is described in the following list:

Function	Action
exit	Performs complete C library termination procedures, terminates the process, and exits with the supplied status code
_exit	Performs "quick" C library termination procedures, terminates the process, and exits with the supplied status code
_cexit	Performs complete C library termination procedures and returns to caller, but does not terminate the process
_c_exit	Performs "quick" C library termination procedures and returns to caller, but does not terminate the process

Return Value

None.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

abort, **atexit**, **exec** functions, **exit**, **onexit**, **spawn** functions, **system**

Description

Gets a character string from the console.

#include <conio.h>	Required only for function declarations

char *cgets(char *_buffer_);

buffer	Storage location for data

Remarks

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 *buffer*. The *buffer* argument must be a pointer to a character array. The first element of the array, *buffer*[0], must contain the maximum length (in characters) of the string to be read. The array must contain 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 (CR-LF) combination 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, *buffer*[1].

Because all DOS editing keys are active when you call **cgets**, pressing F3 repeats the last entry.

Return Value

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

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

getch, getche

Example

```
/* CGETS.C: 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.
 */

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

```
void main()
{
   char buffer[82] = { 80 };  /* Maximum characters in first byte */
   char *result;

   printf( "Input line of text, followed by carriage return:\n");
   result = cgets( buffer );  /* Input a line of text */
   printf( "\nLine length = %d\nText = %s\n", buffer[1], result );
}
```

Output

```
Input line of text, followed by carriage return:
This is some text
Line length = 17
Text = This is some text
```

Description

Chains an interrupt from one handler to another.

#include <dos.h>

void _chain_intr(void(_interrupt _far ***target*** **)( ));**

target	Target interrupt routine

Remarks

The **_chain_intr** routine passes control from one interrupt handler to another. The stack and the registers of the first routine are passed to the second, allowing the second routine to return as if it had been called directly.

The **_chain_intr** routine is generally used when a user-defined interrupt handler begins processing, then chains to the original interrupt handler to finish processing.

Chaining is one of two techniques, listed below, that can be used to transfer control from a new interrupt routine to an old one:

1. Call **_chain_intr** with the interrupt routine as an argument. Do this if your routine is finished and you want the second interrupt routine to terminate the interrupt call.

```
void _interrupt new_int( unsigned _es, unsigned _ds,
   unsigned _di, unsigned _si,... )
{
    ++_di;                  /* Initial processing here  */
    _chain_intr( old_int ); /* New DI passed to old_int */
    --_di;                  /* This is never executed   */
}
```

2. Call the interrupt routine (after casting it to an interrupt function if necessary). Do this if you need to do further processing after the second interrupt routine finishes.

```
void _interrupt new_int( unsigned _es, unsigned _ds,
   unsigned _di, unsigned _si,... )
{
    ++_di;                  /* Initial processing here  */
    (*old_int)();           /* New DI passed to old_int */
    _asm mov _di, di        /* Put real DI from old_int */
                            /*   into _di for return    */
}
```

Note that the real registers set by the old interrupt function are not automatically set to the pseudoregisters of the new routine.

Use the **_chain_intr** function when you do not want to replace the default interrupt handler, but you do need to see its input. An example is a TSR (terminate-and-stay-resident) program that checks all keyboard input for a particular "hot key" sequence.

The **_chain_intr** function should be used only with C functions that have been declared with type **_interrupt**. The **_interrupt** declaration ensures that the procedure's entry/exit sequence is appropriate for an interrupt handler.

Return Value

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

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_dos_getvect, _dos_keep, _dos_setvect

Description

Changes the current working directory.

#include <direct.h>	Required only for function declarations
#include <errno.h>	Required for **errno** constants

int chdir(char **dirname* **);**

dirname	Path name of new working directory

Remarks

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

This function can change the current working directory on any drive; it cannot be used to change the default drive itself. 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");
```

Notice that you must place two backslashes (\\ ) in a C string in order to represent a single backslash (\ ); the backslash is the escape character for C strings and therefore requires special handling.

This function call has no apparent immediate effect. However, when the **_chdrive** function is called to change the default drive to C:, the current working directory becomes C:\TEMP.

In OS/2 protected mode, the current working directory is local to a process rather than system-wide. When a process terminates, the current working directory is restored to its original value. Under DOS, the new directory set by the program becomes the new current working directory.

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 which case **errno** is set to **ENOENT**, indicating that the specified path name could not be found.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also _dos_setdrive, mkdir, rmdir, system

Example

```
/* CHGDIR.C: This program uses the chdir function to verify that a
 * given directory exists. Under real mode that directory also becomes
 * the current directory. Under protected mode, it is only the default
 * directory for the current process.
 */

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

void main( int argc, char *argv[] )
{
   if( chdir( argv[1] )   )
      printf( "Unable to locate the directory: %s\n", argv[1] );
   else
      system( "dir *.c" );
}
```

Output

```
[C:\LIBREF] chgdir \tmp

 The volume label in drive C is OS2.
 Directory of C:\TMP

DUP      C        232   4-18-89  11:18a
TEST     C        713   4-07-88   2:49p
     2 File(s)   14155776 bytes free
```

Description

Changes the current working drive.

#include <direct.h>	Required only for function declarations

int _chdrive(int *drive* **);**

drive	Number of new working drive

Remarks

The **_chdrive** function changes the current working drive to the drive specified by *drive*. The *drive* argument uses an integer to specify the new working drive (1=A, 2=B, etc.).

This function changes only the working drive; the **chdir** function changes the working directory.

In OS/2 protected mode, the working drive is local to a process rather than system-wide. When a process terminates, the working drive is restored to its original value. Under DOS, the new drive set by the program becomes the new working drive.

Return Value

The **_chdrive** function returns a value of 0 if the working drive is successfully changed. A return value of –1 indicates an error.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

chdir, _dos_setdrive, _fullpath, _getcwd, _getdrive, mkdir, rmdir, system

Example

```
/* GETDRIVE.C illustrates drive functions including:
 *      _getdrive       _chdrive        _getdcwd
 */

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

```
void main()
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];

   /* Save current drive. */
   curdrive = _getdrive();

   printf( "Available drives are: \n" );

   /* If we can switch to the drive, it exists. */
   for( drive = 1; drive <= 26; drive++ )
      if( !_chdrive( drive ) )
         printf( "%c: ", drive + 'A' - 1 );

   while( 1 )
   {
      printf( "\nType drive letter to check or ESC to quit: " );
      ch = getch();
      if( ch == 27 )
         break;
      if( isalpha( ch ) )
         putch( ch );
      if( _getdcwd( toupper( ch ) - 'A' + 1, path, _MAX_PATH ) != NULL )
         printf( "\nCurrent directory on that drive is %s\n", path );
   }

   /* Restore original drive. This is only necessary for DOS. Under OS/2
    * the current drive of the calling process is always restored.
    */
   _chdrive( curdrive );
   printf( "\n" );
}
```

Output

```
Available drives are:
A: B: C:
Type drive letter to check or ESC to quit: q
Type drive letter to check or ESC to quit: a
Current directory on that drive is A:\

Type drive letter to check or ESC to quit: c
Current directory on that drive is C:\LIBREF

Type drive letter to check or ESC to quit:
```

Description

Changes the file-permission settings.

#include <sys\types.h>

#include <sys\stat.h>

#include <errno.h>

#include <io.h>	Required only for function declarations

int chmod(char ***filename*****, int** *pmode* **);**

filename	Path name of existing file
pmode	Permission setting for file

Remarks

The **chmod** function changes the permission setting of the file specified by *filename*. 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 read-only. Under DOS and OS/2, 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **access, creat, fstat, open, stat**

Example

```
/* CHMOD.C: This program uses chmod to change the mode of a file to
 * read-only. It then attempts to modify the file.
 */

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

void main()
{
   /* Make file read-only: */
   if( chmod( "CHMOD.C", S_IREAD ) == -1 )
      perror( "File not found\n" );
   else
      printf( "Mode changed to read-only\n" );
   system( "echo /* End of file */ >> CHMOD.C" );

   /* Change back to read/write: */
   if( chmod( "CHMOD.C", S_IWRITE ) == -1 )
      perror( "File not found\n" );
   else
      printf( "Mode changed to read/write\n" );
}
```

Output

```
Mode changed to read-only
Access denied
Mode changed to read/write
```

Description Changes the file size.

#include <io.h>	Required only for function declarations
#include <errno.h>	

int chsize(int *handle*, **long** *size* **);**

handle	Handle referring to open file
size	New length of file in bytes

Remarks 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 is lost.

In DOS, the directory update is done when a file is closed. Consequently, while a program is running, requests to determine the amount of free disk space may receive inaccurate results.

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 (OS/2 and DOS versions 3.0 and later only).
EBADF	Specified file is read-only or an invalid file handle.
ENOSPC	No space is left on device.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **close, creat, open**

Example

```
/* CHSIZE.C: This program uses filelength to report the size of a
 * file before and after modifying it with chsize.
 */
```

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

void main()
{
   int fh, result;
   unsigned int nbytes = BUFSIZ;

   /* Open a file */
   if( (fh = open( "data", O_RDWR | O_CREAT, S_IREAD | S_IWRITE )) != -1 )
   {
      printf( "File length before: %ld\n", filelength( fh ) );
      if( chsize( fh, 329678 ) == 0 )
         printf( "Size successfully changed\n" );
      else
         printf( "Problem in changing the size\n" );
      printf( "File length after:  %ld\n", filelength( fh ) );
      close( fh );
   }
}
```

Output

```
File length before: 0
Size successfully changed
File length after:  329678
```

Description Gets and clears the floating-point status word.

#include <float.h>

unsigned int _clear87(void);

Remarks 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**.

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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_control87, _status87**

Example

```
/* CLEAR87.C: This program creates various floating-point problems,
 * then uses _clear87 to report on these problems.
 */

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

void main()
{
   double a = 1e-40, b;
   float x, y;

   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() );
}
```

Output

```
Status: 0000 - clear
Status: 0030 - inexact, underflow
Status: 0002 - denormal
```

Description

Resets the error indicator for a stream.

#include <stdio.h>

void clearerr(FILE ****stream* **);

stream	Pointer to **FILE** structure

Remarks

The **clearerr** function resets the error indicator and end-of-file indicator for *stream*. 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**, **fseek**, **fsetpos**, or **rewind** is called.

Return Value

None.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

eof, feof, ferror, perror

Example

```
/* CLEARERR.C: This program creates an error on the standard input
 * stream, then clears it so that future reads won't fail.
 */

#include <stdio.h>

void main()
{
   int c;

   /* Create an error by writing to standard input. */
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      clearerr( stdin );
   }
```

```
   /* See if read causes an error. */
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      clearerr( stdin );
   }
}
```

Output

```
Write error: Error 0
Will input cause an error? n
```

Description Clears the specified area of the screen.

#include <graph.h>

void _far _clearscreen(short *area* **);**

area Target area

Remarks 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 None.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_getbkcolor, _setbkcolor**

Example

```
/* CLRSCRN.C */
#include <conio.h>
#include <graph.h>
#include <stdlib.h>

void main()
{
   short xhalf, yhalf, xquar, yquar;
   struct videoconfig vc;
```

```
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _getvideoconfig( &vc );

   xhalf = vc.numxpixels / 2;
   yhalf = vc.numypixels / 2;
   xquar = xhalf / 2;
   yquar = yhalf / 2;

   _setviewport( Ø, Ø, xhalf - 1, yhalf - 1 );
   _rectangle( _GBORDER, Ø,  Ø, xhalf - 1, yhalf - 1 );
   _ellipse( _GFILLINTERIOR, xquar / 4, yquar / 4,
                       xhalf - (xquar / 4), yhalf - (yquar / 4) );
   getch();
   _clearscreen( _GVIEWPORT );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Calculates the time used by the calling process.

#include <time.h>

clock_t clock(void);

Remarks 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 **CLOCKS_PER_SEC** constant.

In other words, the **clock** function returns the number of processor timer ticks that have elapsed. A timer tick is approximately equal to 1/**CLOCKS_PER_SEC** seconds.

Return Value The **clock** function returns the product of the time in seconds and the value of the **CLOCKS_PER_SEC** constant. If the processor time is not available, the function returns the value –1, cast as **clock_t**.

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

In both DOS and OS/2, **clock** returns the time elapsed since the process started. This may not be equal to the actual processor time used by the process.

In previous versions of Microsoft C, the **CLOCKS_PER_SEC** constant was called **CLK_TCK**.

See Also **difftime, time**

Example

```
/* CLOCK.C: This example prompts for how long the program is to run and
 * then continuously displays the elapsed time for that period.
 */

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

void sleep( clock_t wait );

void main()
{
   long    i = 600000L;
   clock_t start, finish;
   double  duration;
```

```
   /* Delay for a specified time. */
   printf( "Delay for three seconds\n" );
   sleep( (clock_t)3 * CLOCKS_PER_SEC );
   printf( "Done!\n" );

   /* Measure the duration of an event. */
   printf( "Time to do %ld empty loops is ", i );
   start = clock();
   while( i-- )
      ;
   finish = clock();
   duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf( "%2.1f seconds\n", duration );
}

/* Pauses for a specified number of microseconds. */
void sleep( clock_t wait )
{
    clock_t goal;

    goal = wait + clock();
    while( goal > clock() )
        ;
}
```

Output

```
Delay for five seconds
Done!
Time to do 900000 empty loops is 2.0 seconds
```

Description Closes a file.

#include <io.h> Required only for function declarations

#include <errno.h>

int close(int *handle* **);**

handle Handle referring to open file

Remarks 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

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

Example

```
/* OPEN.C: This program uses open to open a file named OPEN.C for input
 * and a file named OPEN.OUT for output. The files are then closed.
 */

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

void main()
{
   int fh1, fh2;
   fh1 = open( "OPEN.C", O_RDONLY );
   if( fh1 == -1 )
      perror( "open failed on input file" );
   else
   {
      printf( "open succeeded on input file\n" );
      close( fh1 );
   }
```

```
    fh2 = open( "OPEN.OUT", 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" );
       close( fh2 );
    }
}
```

Output

```
open succeeded on input file
open succeeded on output file
```

Description

Gets and sets the floating-point control word.

#include <float.h>

unsigned int _control87(unsigned int *new*, **unsigned int** *mask* **);**

new	New control-word bit values
mask	Mask for new control-word bits to set

Remarks

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.

The possible values for the mask constant (*mask*) and new control values (*new*) are shown in Table R.1.

Table R.1 Hex Values

Mask	Hex Value	Constant	Hex Value
MCW_EM (Interrupt exception)	0x003F		
		EM_INVALID	0x0001
		EM_DENORMAL	0x0002
		EM_ZERODIVIDE	0x0004
		EM_OVERFLOW	0x0008
		EM_UNDERFLOW	0x0010
		EM_INEXACT	0x0020

Table R.1 *(continued)*

Mask	Hex Value	Constant	Hex Value
MCW_IC (Infinity control)	0x1000		
		IC_AFFINE	0x1000
		IC_PROJECTIVE	0x0000
MCW_RC (Rounding control)	0x0C00		
		RC_CHOP	0x0C00
		RC_UP	0x0800
		RC_DOWN	0x0400
		RC_NEAR	0x0000
MCW_PC (Precision control)	0x0300		
		PC_24 (24 bits)	0x0000
		PC_53 (53 bits)	0x0200
		PC_64 (64 bits)	0x0300

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.**

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_clear87, _status87

Example

```
/* CNTRL87.C: This program uses _control87 to output the control word,
 * set the precision to 24 bits, and reset the status to the default.
 */

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

```
void main()
{
   double a = 0.1;

   /* Show original control word and do calculation. */
   printf( "Original: 0x%.4x\n", _control87( 0, 0 ) );
   printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

   /* Set precision to 24 bits and recalculate. */
   printf( "24-bit:   0x%.4x\n", _control87( PC_24, MCW_PC ) );
   printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

   /* Restore to default and recalculate. */
   printf( "Default:  0x%.4x\n", _control87( CW_DEFAULT, 0xffff ) );
   printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );
}
```

Output

```
Original: 0x1332
0.1 * 0.1 = 1.000000000000000e-002
24-bit:   0x1332
0.1 * 0.1 = 9.999999776482582e-003
Default:  0x1032
0.1 * 0.1 = 1.000000000000000e-002
```

Description

Calculate the cosine (**cos** and **cosl**) or hyperbolic cosine (**cosh** and **coshl**).

#include <math.h>

double cos(double *x* **);**

double cosh(double *x* **);**

long double cosl(long double *x* **);**

long double coshl(long double *x* **);**

x	Angle in radians

Remarks

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

The **cosl** and **coshl** functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

If *x* is large, a partial loss of significance in the result may occur in a call to **cos**, 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.**

Compatibility

cos, cosh

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

cosl, coshl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **acos** functions, **asin** functions, **atan** functions, **matherr**, **sin** functions, **tan** functions

Example

```
/* SINCOS.C: This program displays the sine, hyperbolic sine, cosine,
 * and hyperbolic cosine of pi / 2.
 */

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

void main()
{
   double pi = 3.1415926535;
   double x, y;

   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

Output

```
sin( 1.570796 ) = 1.000000
sinh( 1.570796 ) = 2.301299
cos( 1.570796 ) = 0.000000
cosh( 1.570796 ) = 2.509178
```

Description

Formats and prints to the console.

#include <conio.h> Required only for function declarations

int cprintf(char **format* [[, *argument*]] ... **);**

format	Format control string
argument	Optional arguments

Remarks

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 **printf** for a description of the format and arguments.

Note that unlike the **fprintf, printf,** and **sprintf** functions, **cprintf** does not translate linefeed characters into carriage-return–line-feed combinations on output.

Return Value

The **cprintf** function returns the number of characters printed.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

cscanf, fprintf, printf, sprintf, vprintf

Example

```
/* CPRINTF.C: This program displays some variables to the console. */

#include <conio.h>

void main()
{
   int      i = -16, h = 29;
   unsigned u = 62511;
   char     c = 'A';
   char     s[] = "Test";
```

```
   /* Note that console output does not translate \n as
    * standard output does. Use \r\n instead.
    */
   cprintf( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );

}
```

Output

```
-16  001d  62511  A Test
```

Description

Puts a string to the console.

#include <conio.h> Required only for function declarations

int cputs(char ***string*** **);**

string Output string

Remarks

The **cputs** function writes the null-terminated string pointed to by *string* directly to the console. Note that a carriage-return–line-feed (CR-LF) combination is not automatically appended to the string.

Return Value

If successful, **cputs** returns a 0. If the function fails, it returns a nonzero value.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

putch

Example

```
/* CPUTS.C: This program first displays a string to the console. */

#include <conio.h>

void main()
{
   /* String to print at console. Note the \r (return) character. */
   char *buffer = "Hello world (courtesy of cputs)!\r\n";

   cputs( buffer );
}
```

Output

```
Hello world (courtesy of cputs)!
```

Description Creates a new file.

#include <sys\types.h>

#include <sys\stat.h>

#include <errno.h>

#include <io.h> Required only for function declarations

int creat(char **filename*, **int** *pmode* **);**

filename	Path name of new file
pmode	Permission setting

Remarks The **creat** function either creates a new file or opens and truncates an existing file. If the file specified by *filename* 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 of the constants are given, they are joined with the bitwise-OR operator (|). The *pmode* argument is set to one of the following values:

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 DOS and OS/2, it is not possible to give write-only permission. Thus, the **S_IWRITE** and **S_IREAD | S_IWRITE** modes are equivalent. Under 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**).

Note that 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.

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

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **chmod, chsize, close, dup, dup2, open, sopen, umask**

Example

```
/* CREAT.C: This program uses creat to create the file (or truncate the
 * existing file) named data and open it for writing.
 */

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

void main()
{
   int fh;

   fh = creat( "data", S_IREAD | S_IWRITE );
   if( fh == -1 )
      perror( "Couldn't create data file" );
   else
   {
      printf( "Created data file.\n" );
      close( fh );
   }
}
```

Output

```
Created data file.
```

Description Reads formatted data from the console.

#include <conio.h> Required only for function declarations

int cscanf(char ***format*** **[[, *argument*]] ...);**

format	Format-control string
argument	Optional arguments

Remarks The **cscanf** function reads data directly from the console into the locations given by *argument*. The **getche** function is used to read characters. Each optional 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 **scanf** for a description of *format*.

While **cscanf** 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. This may occur when keyboard input is redirected at the operating system command-line level. A return value of 0 means that no fields were assigned.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **cprintf, fscanf, scanf, sscanf**

Example

```
/* CSCANF.C: This program prompts for a string and uses cscanf to read
 * in the response. Then cscanf returns the number of items matched,
 * and the program displays that number.
 */

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

```
void main()
{
   int   result, i[3];

   cprintf( "Enter three integers: ");
   result = cscanf( "%i %i %i", &i[0], &i[1], &i[2] );
   cprintf( "\r\nYou entered " );
   while( result-- )
      cprintf( "%i ", i[result] );
   cprintf( "\r\n" );

}
```

Output

```
Enter three integers: 34 43 987k
You entered 987 43 34
```

Description Converts a time stored as a **time_t** value to a character string.

#include <time.h> Required only for function declarations

char *ctime(const time_t ***timer*** **);**

timer Pointer to stored time

Remarks The **ctime** function converts a time stored as a **time_t** value to a character string. The *timer* 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 newline character (**\n**) and the null character (**'\0'**) occupy the last two positions of the string.

Calls to the **ctime** function modify the single statically allocated buffer used by the **gmtime** and the **localtime** functions. Each call to one of these routines destroys the result of the previous call. The **ctime** function also shares a static buffer with the **asctime** function. Thus, a call to **ctime** destroys the results of any previous call to **asctime, localtime,** or **gmtime.**

Return Value The **ctime** function returns a pointer to the character string result. If *time* represents a date before 1980, **ctime** returns **NULL.**

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **asctime, ftime, gmtime, localtime, time**

Example

```
/* ASCTIME.C: This program places the system time in the long integer aclock,
 * translates it into the structure newtime and then converts it to
 * string form for output, using the asctime function.
 */

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

```
struct tm *newtime;
time_t aclock;

void main()
{
   time( &aclock );                    /* Get time in seconds. */

   newtime = localtime( &aclock );     /* Convert time to struct tm form. */

   /* Print local time as a string. */
   printf( "The current date and time are: %s\n", asctime( newtime ) );
}
```

Output

```
The current date and time are: Thu Jun 15 06:57:59 1989
```

Description

Waits until the child process terminates.

#include <process.h>

int cwait(int ****termstat*, int** *procid***, int** *action* **);**

termstat	Address for termination status code
procid	Process ID of child
action	Action code

Remarks

The **cwait** function suspends the calling process until the specified child process terminates.

If not **NULL**, *termstat* points to a buffer where **cwait** will place the termination-status word and the return code of the terminated child process.

The termination-status word indicates whether or not the child process terminated normally by calling the OS/2 **DosExit** function. (Programs that terminate with **exit** or by "falling off the end of main" use **DosExit** internally.) If the process did terminate normally, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents
High order	Contains the low-order byte of the result code that the child code passed to **DosExit**. The **DosExit** function is called if the child process called **exit** or **_exit**, returned from **main**, or reached the end of **main**. The low-order byte of the result code is either the low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or a random value if the child process reached the end of **main**.
Low order	0 (normal termination).

If the child process terminates without calling **DosExit**, the high-order and low-order bytes of the termination-status word are as follows:

Byte	Contents
High order	0
Low order	Termination code from **DosCWait**:

Code	Meaning
1	Hard-error abort
2	Trap operation
3	**SIGTERM** signal not intercepted

The *procid* argument specifies which child-process termination to wait for. This value is returned by the call to the **spawn** function that started the child process. If the specified child process terminates before **cwait** is called, the function returns immediately.

The *action* argument specifies when the parent process resumes execution, as shown in the following list:

Value	Meaning
WAIT_CHILD	The parent process waits until the specified child process has ended.
WAIT_GRANDCHILD	The parent process waits until the specified child process and all child processes of that child process have ended.

The **WAIT_CHILD** and **WAIT_GRANDCHILD** action codes are defined in PROCESS.H.

Return Value

If the **cwait** function returns after normal termination of the child process, it returns the child's process ID.

If the **cwait** function returns after abnormal termination of the child process, it returns –1 and sets **errno** to **EINTR**.

Otherwise, the **cwait** function returns –1 immediately and sets **errno** to one of the following error codes:

Value	Meaning
ECHILD	No child process exists, or invalid process ID
EINVAL	Invalid action code

Compatibility ☐ ANSI ☐ DOS ■ OS/2 ☐ UNIX ☐ XENIX

Note that the OS/2 **DosExit** function allows programs to return a 16-bit result code. However, the **wait** and **cwait** functions return only the low-order byte of that result code.

See Also **exit, _exit, spawn** functions, **wait**

Example

```
/* CWAIT.C: This program launches several child processes and waits
 * for a specified process to finish.
 */

#define INCL_NOPM
#define INCL_NOCOMMON
#define INCL_DOSPROCESS
#include <os2.h>        /* DosSleep */
#include <process.h>    /* cwait    */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/* Macro to get a random integer within a specified range */
#define getrandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))

struct  CHILD
{
    int     pid;
    char    name[10];
} child[4] = { { 0, "Ann" }, { 0, "Beth"  }, { 0, "Carl" }, { 0, "Dave" } };

void main( int argc, char *argv[] )
{
    int     termstat, pid, c, tmp;

    srand( (unsigned)time( NULL ) );              /* Seed randomizer */
    /* If no arguments, this is the parent. */
    if( argc == 1 )
    {
        /* Spawn children in numeric order. */
        for( c = 0; c < 4; c++ )
            child[c].pid = spawnl( P_NOWAIT, argv[0], argv[0],
                                   child[c].name, NULL );
```

```
        /* Wait for randomly specified child, and respond when done. */
        c = getrandom( 0, 3 );
        printf( "Come here, %s\n", child[c].name );
        cwait( &termstat, child[c].pid, WAIT_CHILD );
        printf( "Thank you, %s\n", child[c].name );
    }

    /* If there are arguments, this must be a child. */
    else
    {
        /* Delay for a period determined by process number. */
        DosSleep( (argv[1][0] - 'A' + 1) * 1000L );
        printf( "Hi, dad. It's %s.\n", argv[1] );
    }
}
```

Output

```
Come here, Carl
Hi, dad. It's Ann.
Hi, dad. It's Beth.
Hi, dad. It's Carl.
Thank you, Carl
Hi, dad. It's Dave.
```

Description

Convert between IEEE double value and Microsoft (MS) binary double value.

#include <math.h>

int dieeetomsbin(double **src8*, **double** **dst8* **);**

int dmsbintoieee(double **src8*, **double** **dst8* **);**

src8	Buffer containing value to convert
dst8	Buffer to store converted value

Remarks

The **dieeetomsbin** routine converts a double-precision number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft (MS) binary format. The routine **dmsbintoieee** converts a double-precision number in MS 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 MS 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*.

These routines do not handle IEEE NANs ("not a number") and infinities. IEEE denormals are treated as 0 in the conversions.

Return Value

These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

fieeetomsbin, fmsbintoieee

difftime

Description Finds the difference between two times.

#include <time.h>	Required only for function declarations

double difftime(time_t *timer1*, **time_t** *timer0* **);**

timer0	Beginning time
timer1	Ending time

Remarks The **difftime** function computes the difference between the supplied time values, *timer0* and *timer1*.

Return Value The **difftime** function returns, in seconds, the elapsed time from *timer0* to *timer1*. The value returned is a double-precision number.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **time**

Example

```
/* DIFFTIME.C: This program calculates the amount of time needed to
 * do a floating-point multiply 50000 times.
 */

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

void main()
{
   time_t   start, finish;
   unsigned loop;
   double   result, elapsed_time;

   printf( "This program will do a floating point multiply 50000 times\n" );
   printf( "Working...\n" );

   time( &start );
   for( loop = 0; loop < 50000L; loop++ )
      result = 3.63 * 5.27;
   time( &finish );
```

```
   elapsed_time = difftime( finish, start );
   printf( "\nProgram takes %6.2f seconds.\n", elapsed_time );
}
```

Output

```
This program will do a floating point multiply 50000 times
Working...

Program takes   4.00 seconds.
```

Description Disables interrupts.

#include <dos.h>

void _disable(void);

Remarks The **_disable** routine disables interrupts by executing an 8086 **CLI** machine instruction. Use **_disable** before modifying an interrupt vector.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_enable**

Description

Sets the cursor toggle for graphics functions.

#include <graph.h>

short _far _displaycursor(short *toggle* **);**

toggle Cursor state

Remarks

Upon entry into each graphic routine, the screen cursor is turned off. The **_displaycursor** function determines whether the cursor will 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.

Return Value

The function returns the previous value of *toggle*. There is no error return.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_gettextcursor, _settextcursor

Example

```
/* DISCURS.C: This program changes the cursor shape using _gettextcursor
 * and _settextcursor, and hides the cursor using _displaycursor.
 */

#include <conio.h>
#include <graph.h>

void main()
{
   short oldcursor;
   short newcursor = 0x007;        /* Full block cursor */

   /* Save old cursor shape and make sure cursor is on */
   oldcursor = _gettextcursor();
   _clearscreen( _GCLEARSCREEN );
   _displaycursor( _GCURSORON );
   _outtext( "\nOld cursor shape: " );
   getch();

   /* Change cursor shape */
   _outtext( "\nNew cursor shape: " );
   _settextcursor( newcursor );
   getch();
```

```
    /* Restore original cursor shape */
    _outtext( "\n" );
    _settextcursor( oldcursor );
}
```

Description Computes the quotient and the remainder of two integer values.

#include <stdlib.h>

div_t div(int *numer*, int *denom*);

numer	Numerator
denom	Denominator

Remarks The **div** function divides *numer* by *denom*, computing the quotient and the remainder. The **div_t** structure contains the following elements:

Element	Description
int quot	Quotient
int rem	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.

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **ldiv**

Example

```
/* DIV.C: This example 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.
 */

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

```
void main( 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 );
}
```

Output

```
[C:\LIBREF] div 876 13
x is 876, y is 13
The quotient is 67, and the remainder is 5
```

Description Allocates a block of memory, using DOS service 0x48.

#include <dos.h>

#include <errno.h>

unsigned _dos_allocmem(unsigned *size*, **unsigned** **seg* **);**

size	Block size to allocate
seg	Return buffer for segment descriptor

Remarks The **_dos_allocmem** function uses DOS service 0x48 to allocate 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 *seg* points to. If the request cannot be satisfied, the maximum possible size (in paragraphs) is returned in this word instead.

Return Value If successful, the **_dos_allocmem** returns 0. Otherwise, it returns the DOS error code and sets **errno** to **ENOMEM**, indicating insufficient memory or invalid arena (memory area) headers.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **alloca, calloc** functions, **_dos_freemem, _dos_setblock, halloc, malloc** functions

Example

```
/* DALOCMEM.C: This program allocates 20 paragraphs of memory, increases
 * the allocation to 40 paragraphs, and then frees the memory space.
 */

#include <dos.h>
#include <stdio.h>

void main()
{
   unsigned segment;
   unsigned maxsize;
```

```
   /* 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" );
}
```

Output

```
allocation successful
allocation increase successful
free memory successful
```

Description

Closes a file using system call INT 0x3E.

#include <dos.h>

#include <errno.h>

unsigned _dos_close(int *handle* **);**

handle	Target file handle

Remarks

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 DOS error code and sets **errno** to **EBADF**, indicating an invalid file handle.

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

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

close, **creat**, **_dos_creat** functions, **_dos_open**, **_dos_read**, **_dos_write**, **dup**, **open**

Example

```
/* DOPEN.C: This program uses DOS I/O functions to open and close a file. */

#include <fcntl.h>
#include <stdio.h>
#include <dos.h>

void main()
{
   int fh;

   /* Open file with _dos_open function */
   if( _dos_open( "data1", O_RDONLY, &fh ) != 0 )
      perror( "Open failed on input file\n" );
   else
      printf( "Open succeeded on input file\n" );
```

```
   /* Close file with _dos_close function */
   if( _dos_close( fh ) != 0 )
      perror( "Close failed\n" );
   else
      printf( "File successfully closed\n" );
}
```

Output

```
Open succeeded on input file
File successfully closed
```

Description

Create a new file.

#include <dos.h>

#include <errno.h>

unsigned _dos_creat(char ****filename*, unsigned** *attrib***, int ****handle* **);**

unsigned _dos_creatnew(char ****filename*, unsigned** *attrib***, int ****handle* **);**

filename	File path name
attrib	File attributes
handle	Handle return buffer

Remarks

The **_dos_creat** and **_dos_creatnew** routines create and open a new file named *filename*; this new file has the access attributes specified in the *attrib* argument. The new file's handle is copied into the integer location pointed to by *handle.* 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 INT 0x3C, and the **_dos_creatnew** routine uses system call INT 0x5B. If the file already exists, **_dos_creat** erases its contents and leaves its attributes unchanged; however, the **_dos_creatnew** routine fails if the file already exists.

Return Value

If successful, both routines return 0. Otherwise, they return the DOS error code and set **errno** to one of the following values:

Constant	Meaning
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)
EMFILE	Too many open file handles
ENOENT	Path or file not found

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* DCREAT.C: 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.
 */

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

void main()
{
   int fh1, fh2;
   int result;

   if( _dos_creat( "data", _A_NORMAL, &fh1 ) != 0 )
      printf( "Couldn't create data file\n" );
   else
   {
      printf( "Created data file.\n" );

      /* If _dos_creat is successful, the _dos_creatnew call
       * will fail since the file exists
       */
      if( _dos_creatnew( "data", _A_RDONLY, &fh2 ) != 0 )
         printf( "Couldn't create data file\n" );
      else
      {
         printf( "Created data file.\n" );
         _dos_close( fh2 );
      }
      _dos_close( fh1 );
   }
}
```

Output

```
Created data file.
Couldn't create data file
```

Description Find the file with the specified attributes or find the next file with the specified attributes.

#include <dos.h>

#include <errno.h>

unsigned _dos_findfirst(char ***filename*****, unsigned** *attrib***, struct find_t** ***fileinfo*);**

unsigned _dos_findnext(struct find_t ***fileinfo*);**

filename	Target file name
attrib	Target attributes
fileinfo	File-information buffer

Remarks The **_dos_findfirst** routine uses system call INT 0x4E to return information about the first instance of a file whose name and attributes match *filename* and *attrib*.

The *filename* argument may use wildcards (* and ?). The *attrib* argument can be any of the following manifest constants:

Constant	Meaning
_A_ARCH	Archive. Set whenever the file is changed, and cleared by the DOS BACKUP command.
_A_HIDDEN	Hidden file. Cannot be found with the DOS DIR command. Returns information about normal files as well as about files with this attribute.
_A_NORMAL	Normal. File can be read or written without restriction.
_A_RDONLY	Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Returns information about normal files as well as about files with this attribute.
_A_SUBDIR	Subdirectory. Returns information about normal files as well as about files with this attribute.
_A_SYSTEM	System file. Cannot be found with the DOS DIR command. Returns information about normal files as well as about files with this attribute.
_A_VOLID	Volume ID. Only one file can have this attribute, and it must be in the root directory.

Multiple constants can be combined (with the OR operator), using the vertical-bar (|) character.

If the *attributes* argument to either of these functions is **_A_RDONLY**, **_A_HIDDEN**, **_A_SYSTEM**, or **_A_SUBDIR**, the function also returns any normal attribute files that match the *filename* argument. That is, a normal file does not have a read-only, hidden, system, or directory attribute.

Information is returned in a **find_t** structure, defined in DOS.H. The **find_t** structure contains the following elements:

Element	Description
char reserved[21]	Reserved for use by 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

The formats for the **wr_time** and **wr_date** elements are in DOS format and are not usable by any other C run-time function. The time format is shown below:

Bits	Contents
0–4	Number of 2-second increments (0–29)
5–10	Minutes (0–59)
11–15	Hours (0–23)

The date format is shown below:

Bits	Contents
0–4	Day of month (1–31)
5–8	Month (1–12)
9–15	Year (relative to 1980)

Do not alter the contents of the buffer between a call to **_dos_findfirst** and a subsequent call to the **_dos_findnext** function. Also, the buffer should not be altered between calls to **_dos_findnext**.

The **_dos_findnext** routine uses system call 0x4F to find the next name, if any, that matches the *filename* and *attrib* arguments specified in a prior call to **_dos_findfirst**. The *fileinfo* argument must point to a structure 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 DOS error code and set **errno** to **ENOENT**, indicating that *filename* could not be matched.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* DFIND.C: This program finds and prints all files in the current directory with
 * the .c extension.
 */

#include <stdio.h>
#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 %ld bytes\n", c_file.name, c_file.size );

   /* find the rest of the .c files */
   while( _dos_findnext( &c_file ) == 0 )
      printf( "File: %s is %ld bytes\n", c_file.name, c_file.size );
}
```

Output

```
Listing of .c files

File: CHDIR.C is 524 bytes
File: SIGFP.C is 2674 bytes
File: MAX.C is 258 bytes
File: CGETS.C is 577 bytes
File: FWRITE.C is 1123 bytes
```

Description Releases a block of memory (INT 0x49).

#include <dos.h>

#include <errno.h>

unsigned _dos_freemem(unsigned *seg* **);**

seg	Block to be released

Remarks The **_dos_freemem** function uses system call 0x49 to release a block of memory previously allocated by **_dos_allocmem.** The *seg* argument is a value returned by a previous call to **_dos_allocmem.** The freed memory may no longer be used by the application program.

Return Value If successful, **_dos_freemem** returns 0. Otherwise, it returns the 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** call) or invalid arena headers.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_dos_allocmem, _dos_setblock, free** functions

Example

```
/* DALOCMEM.C: This program allocates 20 paragraphs of memory, increases
 * the allocation to 40 paragraphs, and then frees the memory space.
 */

#include <dos.h>
#include <stdio.h>

void main()
{
   unsigned segment;
   unsigned maxsize;

   /* 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" );
}
```

Output

```
allocation successful
allocation increase successful
free memory successful
```

Description Gets current system date using system call INT 0x2A.

#include <dos.h>

void _dos_getdate(struct dosdate_t ***date*** **);**

date Current system date

Remarks 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.

The **dosdate_t** structure contains the following elements:

Element	Description
unsigned char day	1–31
unsigned char month	1–12
unsigned int year	1980–2099
unsigned char dayofweek	0–6 (0 = Sunday)

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_dos_gettime, _dos_setdate, _dos_settime, gmtime, localtime, mktime, _strdate, _strtime, time**

Example

```
/* DGTIME.C: This program gets and displays current date and time values. */

#include <stdio.h>
#include <dos.h>

void 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 %02d:%02d\n", time.hour, time.minute );
}
```

Output

```
Today's date is 6-15-1989
The time is 18:07
```

Description Gets disk information using system call INT 0x36.

#include <dos.h>

#include <errno.h>

unsigned _dos_getdiskfree(unsigned *drive*, **struct diskfree_t** **diskspace* **);**

drive	Drive number (default is 0)
diskspace	Buffer to hold disk information

Remarks 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 (defined in DOS.H) pointed to by *diskspace*.

The **struct diskfree_t** structure contains the following elements:

Element	Description
unsigned total_clusters	Total clusters on disk
unsigned avail_clusters	Available clusters on disk
unsigned sectors_per_cluster	Sectors per cluster
unsigned bytes_per_sector	Bytes per sector

Return Value If successful, the function returns 0. Otherwise, it returns a nonzero value and sets **errno** to **EINVAL**, indicating that an invalid drive was specified.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_dos_getdrive, _dos_setdrive**

Example

```
/* DGDISKFR.C: This program displays information about the default disk drive. */

#include <stdio.h>
#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 );
}
```

Output

```
total clusters: 9013
available clusters: 6030
sectors per cluster: 4
bytes per sector: 512
```

Description Gets the current disk drive using system call INT 0x19.

#include <dos.h>

void _dos_getdrive(unsigned ***drive*);**

drive	Current-drive return buffer

Remarks 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.

Return Value None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_dos_getdiskfree, _dos_setdrive, _getdrive**

Example

```
/* DGDRIVE.C: This program prints the letter of the current drive,
 * changes the default drive to A, then returns the number of disk drives.
 */

#include <stdio.h>
#include <dos.h>

void main()
{
   unsigned olddrive, newdrive;
   unsigned number_of_drives;

   /* Print current default drive information */
   _dos_getdrive( &olddrive );
   printf( "The current drive is: %c\n", 'A' + olddrive - 1 );

   /* Set default drive to be drive A */
   printf( "Changing default drive to A\n");
   _dos_setdrive( 1, &number_of_drives );

   /* Get new default drive information and total number of drives */
   _dos_getdrive( &newdrive );
   printf( "The current drive is: %c\n", 'A' + newdrive - 1 );
   printf( "Number of logical drives: %d\n", number_of_drives );
```

```
   /* Restore default drive */
   _dos_setdrive( olddrive, &number_of_drives );
}
```

Output

```
The current drive is: C
Changing default drive to A
The current drive is: A
Number of logical drives: 26
```

Description

Gets the current attributes of a file or directory, using system call INT 0x43.

#include <dos.h>

#include <errno.h>

unsigned _dos_getfileattr(char ***pathname*****, unsigned** ***attrib* **);**

pathname	Full path of target file/directory
attrib	Word to store attributes in

Remarks

The **_dos_getfileattr** routine uses system call 0x43 to obtain the current attributes of the file or directory pointed to by *pathname* . The attributes are copied to the low-order byte of the *attrib* word. Attributes are represented by manifest constants, as described below:

Constant	Meaning
_A_ARCH	Archive. Set whenever the file is changed, or cleared by the DOS BACKUP command.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_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_SUBDIR	Subdirectory.
_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.

Return Value

If successful, the function returns 0. Otherwise, it returns the DOS error code and sets **errno** to **ENOENT**, indicating that the target file or directory could be found.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **access, chmod, _dos_setfileattr, umask**

Example

```
/* DGFILEAT.C: This program creates a file with the specified attributes,
 * then prints this information before changing the file attributes back
 * to normal.
 */

#include <stdio.h>
#include <dos.h>

void main()
{
   unsigned oldattrib, newattrib;
   int fh;

   /* Get and display file attribute */
   _dos_getfileattr( "DGFILEAT.C", &oldattrib );
   printf( "Attribute: 0x%.4x\n", oldattrib );
   if( ( oldattrib & _A_RDONLY ) != 0 )
      printf( "Read only file\n" );
   else
      printf( "Not a read only file.\n" );

   /* Reset file attribute to normal file */
   _dos_setfileattr( "DGFILEAT.C", _A_RDONLY );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );

   /* Restore file attribute */
   _dos_setfileattr( "DGFILEAT.C", oldattrib );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );
}
```

Output

```
Attribute: 0x0020
Not a read only file.
Attribute: 0x0001
Attribute: 0x0020
```

Description

Gets the date and time a file was last written, using system call INT 0x57.

#include <dos.h>

#include <errno.h>

unsigned _dos_getftime(int *handle*, **unsigned** **date*, **unsigned** **time* **);**

handle	Target file
date	Date-return buffer
time	Time-return buffer

Remarks

The **_dos_getftime** routine uses system call 0x57 to get the date and time that the specified file was last written. The file must have been opened with a call to **_dos_open** or **_dos_creat** prior to calling **_dos_getftime**. The date and time are returned in the words pointed to by *date* and *time*. The values appear in the DOS date and time format:

Time Bits	Meaning
0–4	Number of 2-second increments (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 DOS error code and sets **errno** to **EBADF**, indicating that an invalid file handle was passed.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_dos_setftime, fstat, stat**

Example

```
/* DGFTIME.C: This program displays and modifies the date and time
 * fields of a file.
 */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main()
{
                                   /* FEDC BA98 7654 3210          */
   unsigned new_date = 0x184f;    /* 0001 1000 0100 1111  2/15/92 */
   unsigned new_time = 0x48e0;    /* 0100 1000 1110 0000  9:07 AM */
   unsigned old_date, old_time;

   int fh;

   /* Open file with _dos_open function */
   if( _dos_open( "dgftime.obj", O_RDONLY, &fh ) != 0 )
      exit( 1 );

   /* Get file date and time */
   _dos_getftime( fh, &old_date, &old_time );
   printf( "Old date field: 0x%.4x\n", old_date );
   printf( "Old time field: 0x%.4x\n", old_time );
   system( "dir dgftime.obj" );

   /* Modify file date and time */
   if( !_dos_setftime( fh, new_date, new_time ) )
   {
      _dos_getftime( fh, &new_date, &new_time );
      printf( "New date field: 0x%.4x\n", new_date );
      printf( "New time field: 0x%.4x\n", new_time );
      system( "dir dgftime.obj" );

      /* Restore date and time */
      _dos_setftime( fh, old_date, old_time );
   }
   _dos_close( fh );
}
```

Output

```
Old date field: Øx12cf
Old time field: Øx94bb

 Volume in drive C is OS2
 Directory of  C:\LIBREF

DGFTIME  OBJ     3923   6-15-89   6:37p
        1 File(s)  13676544 bytes free

New date field: Øx184f
New time field: Øx48eØ

 Volume in drive C is OS2
 Directory of  C:\LIBREF

DGFTIME  OBJ     3923   2-15-92   9:Ø7a
        1 File(s)  13676544 bytes free
```

Description

Gets the current system time, using system call INT 0x2C.

#include <dos.h>

void _dos_gettime(struct dostime_t ****time*** **);**

time Current system time

Remarks

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.

The **dostime_t** structure contains the following elements:

Element	Description
unsigned char hour	0–23
unsigned char minute	0–59
unsigned char second	0–59
unsigned char hsecond	1/100 second; 0–99

Return Value

None.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_dos_getdate, _dos_setdate, _dos_settime, gmtime, localtime, _strtime

Example

```
/* DGTIME.C: This program gets and displays current date and time values. */

#include <stdio.h>
#include <dos.h>

void 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 %02d:%02d\n", time.hour, time.minute );
}
```

Output

```
Today's date is 6-15-1989
The time is 18:07
```

Description Gets the current value of the interrupt vector, using system call INT 0x35.

#include <dos.h>

void (_interrupt _far *_dos_getvect(unsigned *intnum*))( );

intnum	Target interrupt vector

Remarks The **_dos_getvect** routine uses system call INT 0x35 to get the current value of the interrupt vector specified by *intnum*.

This routine is typically used in conjunction with the **_dos_setvect** function. To replace an interrupt vector, first save the current vector of the interrupt using **_dos_getvect**. Then set the vector to your own interrupt routine with **_dos_setvect**. The saved vector can later be restored, if necessary, using **_dos_setvect**. The user-defined routine may also need the original vector in order to call that vector or chain to it with **_chain_intr**.

Return Value The function returns a far pointer for the *intnum* interrupt to the current handler, if there is one.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_chain_intr, _dos_keep, _dos_setvect**

Description

Installs TSR (terminate-and-stay-resident) programs in memory, using system call INT 0x31.

#include <dos.h>

void _dos_keep(unsigned *retcode*, **unsigned** *memsize* **);**

retcode	Exit status code
memsize	Allocated resident memory (in 16-byte paragraphs)

Remarks

The **_dos_keep** routine installs TSRs (terminate-and-stay-resident programs) in memory, using system call INT 0x31.

The routine first exits the calling process, leaving it in memory. It then 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* 16-byte paragraphs. Any excess memory is returned to the system.

The **_dos_keep** function calls the same internal routines called by **exit**. It therefore takes the following actions:

- Calls **atexit** and **onexit** if defined.
- Flushes all file buffers.
- Restores interrupt vectors replaced by the C start-up code. The primary one is interrupt 0 (divide by zero). If the emulator math library is used and there is no coprocessor, interrupts 0x34 through 0x3D are restored. If there is a coprocessor, interrupt 2 is restored.

The **_dos_keep** function does not automatically close files; you should do this specifically unless you want files opened by the TSR installation code to remain open for the TSR.

Do not use the emulator math library in TSRs unless you are familiar with the C start-up code and the coprocessor. Use the alternate math package (not supplied with Microsoft QuickC) if the TSR must do floating-point math.

Do not run programs that use **_dos_keep** from inside the Microsoft Programmer's WorkBench environment, since doing so causes subsequent memory problems. The **_dos_keep** function terminates the program when executed in the Programmer's WorkBench environment.

Return Value

None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_cexit, _chain_intr, _dos_getvect, _dos_setvect, _exit**

Description

Opens a file, using system call INT 0x3D.

#include <dos.h>	
#include <errno.h>	
#include <fcntl.h>	Access mode constants
#include <share.h>	Sharing mode constants

unsigned _dos_open(char **filename*, **unsigned** *mode*, **int** **handle* **);**

filename	Path to an existing file
mode	Permissions
handle	Pointer to integer

Remarks

The **_dos_open** routine uses system call INT 0x3D to open the existing file pointed to by *filename*. The handle for the opened file is copied into the integer pointed to by *handle*. The *mode* argument specifies the file's access, sharing, and inheritance modes by combining (with the OR operator) manifest constants from the three groups shown below. At most, one access mode and one sharing mode can 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_DENYNO	Sharing	Deny neither
O_NOINHERIT	Inheritance by the child process	File is not inherited

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

Return Value If successful, the function returns 0. Otherwise, it returns the DOS error code and sets **errno** to one of the following manifest constants:

Constant	Meaning
EACCES	Access denied (possible reasons include specifying a directory or volume ID for *filename*, or opening a read-only file for write access)
EINVAL	Sharing mode specified when file sharing not installed, or access-mode value is invalid
EMFILE	Too many open file handles
ENOENT	Path or file not found

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_dos_close, _dos_read, _dos_write**

Example

```
/* DOPEN.C: This program uses DOS I/O functions to open and close a file. */

#include <fcntl.h>
#include <stdio.h>
#include <dos.h>

void main()
{
   int fh;

   /* Open file with _dos_open function */
   if( _dos_open( "data1", O_RDONLY, &fh ) != 0 )
      perror( "Open failed on input file\n" );
   else
      printf( "Open succeeded on input file\n" );

   /* Close file with _dos_close function */
   if( _dos_close( fh ) != 0 )
      perror( "Close failed\n" );
   else
      printf( "File successfully closed\n" );
}
```

Output

```
Open succeeded on input file
File successfully closed
```

Description Reads data from a file, using system call INT 0x3F.

#include <dos.h>

unsigned _dos_read(int *handle*, **void _far** **buffer*, **unsigned** *count*,
unsigned **numread*);

handle	File to read
buffer	Buffer to write to
count	Number of bytes to read
numread	Number of bytes actually read

Remarks The **_dos_read** routine uses system call INT 0x3F to read *count* bytes of data from the file specified by *handle*. The routine then copies the data to the buffer pointed to by *buffer*. The integer pointed to by *numread* 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 end-of-file.

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

Return Value If successful, the function returns 0. Otherwise, it returns the DOS error code and sets **errno** to one of the following constants:

Constant	Meaning
EACCES	Access denied (*handle* is not open for read access)
EBADF	File handle is invalid

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_dos_close, _dos_open, _dos_write, read**

Example

```
/* DREAD.C: This program uses the DOS I/O operations to read the contents
 * of a file.
 */
```

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

void 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\n" );
   else
      printf( "Open succeeded on input file\n" );

   /* Read data with _dos_read function */
   _dos_read( fh, buffer, 50, &number_read );
   printf( "First 40 characters are: %.40s\n\n", buffer );

   /* Close file with _dos_close function */
   _dos_close( fh );
}
```

Output

```
Open succeeded on input file
First 40 characters are: /* DREAD.C: This program uses the DOS I/
```

Description

Changes the size of a memory segment, using system call INT 0x4A.

#include <dos.h>

unsigned _dos_setblock(unsigned *size*, **unsigned** *seg*, **unsigned** **maxsize* **);**

size	New segment size
seg	Target segment
maxsize	Maximum-size buffer

Remarks

The **_dos_setblock** routine uses system call INT 0x4A to change the size of *seg*, previously allocated by **_dos_allocmem**, to *size* paragraphs. If the request cannot be satisfied, the maximum possible segment size is copied to the buffer pointed to by *maxsize.*

Return Value

The function returns 0 if successful. If the call fails, it returns the DOS error code and sets **errno** to **ENOMEM**, indicating a bad segment value was passed. A bad segment value is one that does not correspond to a segment returned from a previous **_dos_allocmem** call, or one that contains invalid arena headers.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_dos_allocmem, _dos_freemem, realloc functions

Example

```
/* DALOCMEM.C: This program allocates 20 paragraphs of memory, increases
 * the allocation to 40 paragraphs, and then frees the memory space.
 */

#include <dos.h>
#include <stdio.h>

void main()
{
   unsigned segment;
   unsigned maxsize;
```

```
    /* 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" );
}
```

Output

```
allocation successful
allocation increase successful
free memory successful
```

Description

Sets the current system date, using system call INT 0x2B.

#include <dos.h>

unsigned _dos_setdate(struct dosdate_t **date* **);**

date	New system date

Remarks

The **_dos_setdate** routine uses system call INT 0x2B to set the current system date. The date is stored in the **dosdate_t** structure pointed to by *date*, defined in DOS.H. The **dosdate_t** structure contains the following elements:

Element	Description
unsigned char day	1–31
unsigned char month	1–12
unsigned int year	1980–2099
unsigned char dayofweek	0–6 (0 = Sunday)

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_dos_gettime, _dos_setdate, _dos_settime, gmtime, localtime, mktime, _strdate, _strtime, time

Example

```
/* DSTIME.C: This program changes the time and date values and displays the
 * new date and time values.
 */

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

```
void main()
{
   struct dosdate_t olddate, newdate = { { 4 }, { 7 }, { 1984 } };
   struct dostime_t oldtime, newtime = { { 3 }, { 45 }, { 30 }, { 0 } };
   char   datebuf[40], timebuf[40];

   /* Get current date and time values */
   _dos_getdate( &olddate );
   _dos_gettime( &oldtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );

   /* Modify date and time structures */
   _dos_setdate( &newdate );
   _dos_settime( &newtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );

   /* Restore old date and time */
   _dos_setdate( &olddate );
   _dos_settime( &oldtime );
}
```

Output

```
06/15/89    18:26:09
07/04/84    03:45:30
```

Description

Sets the default drive, using system call INT 0x0E.

#include <dos.h>

void _dos_setdrive(unsigned *drive*, **unsigned** ***numdrives*);**

drive	New default drive
numdrives	Total drives available

Remarks

The **_dos_setdrive** routine uses system call INT 0x0E to set the current default drive to the *drive* argument: 1 = drive A, 2 = drive B, and so on. The *numdrives* argument indicates the total number of drives in the system. If this value is 4, for example, it does not mean the drives are designated A, B, C, and D; it means only that four drives are in the system.

Return Value

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_dos_getdiskfree, _dos_getdrive

Example

```
/* DGDRIVE.C: This program prints the letter of the current drive,
 * changes the default drive to A, then returns the number of disk drives.
 */

#include <stdio.h>
#include <dos.h>

void main()
{
   unsigned olddrive, newdrive;
   unsigned number_of_drives;

   /* Print current default drive information */
   _dos_getdrive( &olddrive );
   printf( "The current drive is: %c\n", 'A' + olddrive - 1 );

   /* Set default drive to be drive A */
   printf( "Changing default drive to A\n");
   _dos_setdrive( 1, &number_of_drives );
```

```
   /* Get new default drive information and total number of drives */
   _dos_getdrive( &newdrive );
   printf( "The current drive is: %c\n", 'A' + newdrive - 1 );
   printf( "Number of logical drives: %d\n", number_of_drives );

   /* Restore default drive */
   _dos_setdrive( olddrive, &number_of_drives );
}
```

Output

```
The current drive is: C
Changing default drive to A
The current drive is: A
Number of logical drives: 26
```

Description

Sets the attributes of the file or directory, using system call INT 0x43.

#include <dos.h>

unsigned _dos_setfileattr(char **pathname*, **unsigned** *attrib* **);**

pathname	Full path of target file/directory
attrib	New attributes

Remarks

The **_dos_setfileattr** routine uses system call INT 0x43 to set the attributes of the file or directory pointed to by *pathname*. The actual attributes are contained in the low-order byte of the *attrib* word. Attributes are represented by manifest constants, as described below:

Constant	Meaning
_A_ARCH	Archive. Set whenever the file is changed, or cleared by the DOS BACKUP command.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_A_NORMAL	Normal. File can be read or written to without restriction.
_A_RDONLY	Read-only. File cannot be opened for writing, and a file with the same name cannot be created.
_A_SUBDIR	Subdirectory.
_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.

Return Value

The function returns 0 if successful. Otherwise, it returns the DOS error code and sets **errno** to one of the following:

Constant	Meaning
EACCES	Access denied; cannot change the volume ID or the subdirectory.
ENOENT	No file or directory matching the target was found.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also _dos_getfileattr

Example

```
/* DGFILEAT.C: This program creates a file with the specified attributes,
 * then prints this information before changing the file attributes back
 * to normal.
 */

#include <stdio.h>
#include <dos.h>

void main()
{
   unsigned oldattrib, newattrib;
   int fh;

   /* Get and display file attribute */
   _dos_getfileattr( "DGFILEAT.C", &oldattrib );
   printf( "Attribute: 0x%.4x\n", oldattrib );
   if( ( oldattrib & _A_RDONLY ) != 0 )
      printf( "Read only file\n" );
   else
      printf( "Not a read only file.\n" );

   /* Reset file attribute to normal file */
   _dos_setfileattr( "DGFILEAT.C", _A_RDONLY );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );

   /* Restore file attribute */
   _dos_setfileattr( "DGFILEAT.C", oldattrib );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );
}
```

Output

```
Attribute: 0x0020
Not a read only file.
Attribute: 0x0001
Attribute: 0x0020
```

Description

Sets the date and time for a file, using system call INT 0x57.

#include <dos.h>

unsigned _dos_setftime(**int** *handle*, **unsigned** *date*, **unsigned** *time* **);**

handle	Target file
date	Date of last write
time	Time of last write

Remarks

The **_dos_setftime** routine uses system call INT 0x57 to set the *date* and *time* at which the file identified by *handle* was last written to. These values appear in the DOS date and time format, described in the following lists:

Time Bits	Meaning
0–4	Number of two-second increments (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 since 1980 (for example, 1989 is stored as 9)

Return Value

If successful, the function returns 0. Otherwise, it returns the DOS error code and sets **errno** to **EBADF**, indicating that an invalid file handle was passed.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also _dos_getftime, fstat, stat

Example

```
/* DGFTIME.C: This program displays and modifies the date and time
 * fields of a file.
 */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main()
{
                                  /* FEDC BA98 7654 3210          */
   unsigned new_date = 0x184f;    /* 0001 1000 0100 1111  2/15/92 */
   unsigned new_time = 0x48e0;    /* 0100 1000 1110 0000  9:07 AM */
   unsigned old_date, old_time;

   int fh;

   /* Open file with _dos_open function */
   if( _dos_open( "dgftime.obj", O_RDONLY, &fh ) != 0 )
      exit( 1 );

   /* Get file date and time */
   _dos_getftime( fh, &old_date, &old_time );
   printf( "Old date field: 0x%.4x\n", old_date );
   printf( "Old time field: 0x%.4x\n", old_time );
   system( "dir dgftime.obj" );

   /* Modify file date and time */
   if( !_dos_setftime( fh, new_date, new_time ) )
   {
      _dos_getftime( fh, &new_date, &new_time );
      printf( "New date field: 0x%.4x\n", new_date );
      printf( "New time field: 0x%.4x\n", new_time );
      system( "dir dgftime.obj" );

      /* Restore date and time */
      _dos_setftime( fh, old_date, old_time );
   }
   _dos_close( fh );
}
```

Output

```
Old date field: 0x12cf
Old time field: 0x94bb

 Volume in drive C is OS2
 Directory of  C:\LIBREF

DGFTIME  OBJ     3923   6-15-89   6:37p
        1 File(s)  13676544 bytes free

New date field: 0x184f
New time field: 0x48e0

 Volume in drive C is OS2
 Directory of  C:\LIBREF

DGFTIME  OBJ     3923   2-15-92   9:07a
        1 File(s)  13676544 bytes free
```

_dos_settime

Description Sets the current system time, using system call INT 0x2D.

#include <dos.h>

unsigned _dos_settime(struct dostime_t ****time* **);**

time New system time

Remarks The **_dos_settime** routine uses system call INT 0x2D to set the current system time to the value stored in the **dostime_t** structure that *time* points to, as defined in DOS.H. The **dostime_t** structure contains the following elements:

Element	Description
unsigned char hour	0–23
unsigned char minute	0–59
unsigned char second	0–59
unsigned char hsecond	Hundredths of a second; 0–99

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.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_dos_getdate, _dos_gettime, _dos_setdate, gmtime, localtime, mktime, _strdate, _strtime**

Example

```
/* DSTIME.C: This program changes the time and date values and displays the
 * new date and time values.
 */

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

```
void main()
{
   struct dosdate_t olddate, newdate = { { 4 }, { 7 }, { 1984 } };
   struct dostime_t oldtime, newtime = { { 3 }, { 45 }, { 30 }, { 0 } };
   char   datebuf[40], timebuf[40];

   /* Get current date and time values */
   _dos_getdate( &olddate );
   _dos_gettime( &oldtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );

   /* Modify date and time structures */
   _dos_setdate( &newdate );
   _dos_settime( &newtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );

   /* Restore old date and time */
   _dos_setdate( &olddate );
   _dos_settime( &oldtime );
}
```

Output

```
06/15/89    18:26:09
07/04/84    03:45:30
```

Description

Sets the current value of the interrupt vector, using system call INT 0x25.

#include <dos.h>

void _dos_setvect(unsigned *intnum*, **void(_interrupt _far** **handler*)());

intnum	Target-interrupt vector
handler	Interrupt handler for which to assign *intnum*

Remarks

The **_dos_setvect** routine uses system call INT 0x25 to set the current value of the interrupt vector *intnum* to the function pointed to by *handler*. 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. For example, if *handler* is an assembler function, it must be a **far** routine that returns with an **IRET** instead of a **RET**.

The **interrupt** attribute indicates that the function is an interrupt handler. The compiler generates appropriate entry and exit sequences for the interrupt-handling function, including saving and restoring all registers and executing an **IRET** instruction to return.

The **_dos_setvect** routine is generally used with the **_dos_getvect** function. To replace an interrupt vector, first save the current vector of the interrupt using **_dos_getvect**. Then set the vector to your own interrupt routine with **_dos_setvect**. The saved vector can later be restored, if necessary, using **_dos_setvect**. The user-defined routine may also need the original vector in order to call it or to chain to it with **_chain_intr**.

Registers and Interrupt Functions

When you call an interrupt function, the DS register is initialized to the C data segment. This allows you to access global variables from within an interrupt function.

In addition, all registers except SS are saved on the stack. You can access these registers within the function if you declare a function parameter list containing a formal parameter for each saved register. The following example illustrates such a declaration:

```
void _interrupt _far int_handler( unsigned _es, unsigned _ds,
                                  unsigned _di, unsigned _si,
                                  unsigned _bp, unsigned _sp,
                                  unsigned _bx, unsigned _dx,
                                  unsigned _cx, unsigned _ax,
                                  unsigned _ip, unsigned _cs,
                                  unsigned _flags )
{
.
.
.
}
```

The formal parameters must appear in the opposite order from which they are pushed onto the stack. You can omit parameters from the end of the list in a declaration, but not from the beginning. For example, if your handler needs to use only DI and SI, you must still provide ES and DS, but not necessarily BX or DX.

You can pass additional arguments if your interrupt handler will be called directly from C rather than by an INT instruction. To do this, you must declare all register parameters and then declare your parameter at the end of the list.

The compiler always saves and restores registers in the same, fixed order. Thus, no matter what names you use in the formal parameter list, the first parameter in the list refers to ES, the second refers to DS, and so on. If your interrupt routines will use in-line assembler, you should distinguish the parameter names so that they will not be the same as the real register names.

If you change any of the register parameters of an interrupt function while the function is executing, the corresponding register contains the changed value when the function returns. For example:

```
void _interrupt _far int_handler( unsigned _es, unsigned _ds,
                                  unsigned _di, unsigned _si )
{
    _di = -1;
}
```

This code causes the DI register to contain –1 when the *handler* function returns. It is not a good idea to modify the values of the parameters representing the IP and CS registers in interrupt functions. If you must modify a particular flag (such as the carry flag for certain DOS and BIOS interrupt routines), use the OR operator (|) so that other bits in the flag register are not changed.

When an interrupt function is called by an INT instruction, the interrupt-enable flag is cleared. If your interrupt function needs to do significant processing, you should use the **_enable** function to set the interrupt flag so that interrupts can be handled.

Precautions for Interrupt Functions

Since DOS is not reentrant (a DOS interrupt cannot be called from inside a DOS interrupt), it is usually not safe to call from inside an interrupt function any standard library function that calls DOS INT 21H. Similar precautions apply to many BIOS functions. Functions that rely on INT 21H calls include I/O functions and the **_dos** family of functions. Functions that rely on the machine's BIOS include graphics functions and the **_bios** family of functions. It is usually safe to use functions that do not rely on INT 21H or BIOS, such as string-handling functions. Before using a standard library function in an interrupt function, be sure that you are familiar with the action of the library function.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_chain_intr**, **_dos_getvect**, **_dos_keep**

Description

Writes a buffer to a file, using system call INT 0x40.

#include <dos.h>

unsigned _dos_write(int *handle*, **void _far** **buffer*, **unsigned** *count*,
unsigned **numwrt* **);**

handle	File to write to
buffer	Buffer to write from
count	Number of bytes to write
numwrt	Number of bytes actually written

Remarks

The **_dos_write** routine uses system call INT 0x40 to write data to the file that *handle* references; *count* bytes of data from the buffer to which *buffer* points are written to the file. The integer pointed to by *numwrt* will be the number of bytes actually written, which may be less than the number requested.

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

Return Value

If successful, the function returns 0. Otherwise, it returns the DOS error code and sets **errno** to one of the following manifest constants:

Constant	Meaning
EACCES	Access denied (*handle* references a file not open for write access)
EBADF	Invalid file handle

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

_dos_close, _dos_open, _dos_read, write

Example

```
/* DWRITE.C: This program uses DOS I/O functions to write to a file. */

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

```
void main()
{
   char out_buffer[] = "Hello";
   int  fh;
   unsigned n_written;

   /* Open file with _dos_creat function */
   if( _dos_creat( "data", _A_NORMAL, &fh ) == 0 )
   {
      /* Write data with _dos_write function */
      _dos_write( fh, out_buffer, 5, &n_written );
      printf( "Number of characters written: %d\n", n_written );

      _dos_close( fh );
      printf( "Contents of file are:\n" );
      system( "type data" );
   }
}
```

Output

```
Number of characters written: 5
Contents of file are:
Hello
```

Description

Gets register values returned by INT 0x59.

#include <dos.h>

int dosexterr(struct DOSERROR **errorinfo* **);**

errorinfo	Extended DOS error information

Remarks

The **dosexterr** function obtains the extended error information returned by the DOS system call INT 0x59 and stores the values in the structure pointed to by *errorinfo.* This function is useful when making system calls under DOS versions 3.0 or later, which offer extended error handling.

The structure type **DOSERROR** is defined in DOS.H. The **DOSERROR** structure contains the following elements:

Element	Description
int exterror	AX register contents
char class	BH register contents
char action	BL register contents
char locus	CH register contents

Giving a **NULL** pointer argument causes **dosexterr** to return the value in AX without filling in the structure fields. See *MS-DOS Encyclopedia* (Duncan, ed.; Redmond, WA: Microsoft Press, 1988) or *Programmer's PC Sourcebook* (Hogan; Redmond, WA: Microsoft Press, 1988) 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).

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

The **dosexterr** function should be used only under DOS versions 3.0 or later.

See Also **perror**

Example

```
/* DOSEXERR.C: This program tries to open the file test.dat. If the
 * attempted open operation fails, the program uses dosexterr to display
 * extended error information.
 */

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

void main()
{
   struct DOSERROR doserror;
   int fd;

   /* Attempt to open a non-existent file */
   if( (fd = open( "NOSUCHF.ILE", 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\n" );
      close( fd );
   }
}
```

Output

```
Error: 2  Class: 8  Action: 3  Locus: 2
```

Description Create a second handle for an open file (**dup**), or reassign a file handle (**dup2**).

#include <io.h	Required only for function declarations

int dup(int *handle* **);**

int dup2(int *handle1*, **int** *handle2* **);**

handle, handle1	Handle referring to open file
handle2	Any handle value

Remarks 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. 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)

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **close, creat, open**

Example

```
/* DUP.C: 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.
 */

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

```
void main()
{
   int old;
   FILE *new;

   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\r\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'\r\n" );

   /* Flush stdout stream buffer so it goes to correct file */
   fflush( stdout );
   fclose( new );

   /* Restore original stdout */
   dup2( old, 1 );
   puts( "This goes to stdout\n" );
   puts( "The file 'data' contains:" );
   system( "type data" );
}
```

Output

```
This goes to stdout first
This goes to stdout

The file 'data' contains:
This goes to file 'data'
```

Description

Converts a **double** number to a string.

#include <stdlib.h>	Required only for function declarations

char *ecvt(double *value*, **int** *count*, **int** **dec*, **int** **sign* **);**

value	Number to be converted
count	Number of digits stored
dec	Stored decimal-point position
sign	Sign of converted number

Remarks

The **ecvt** function converts a floating-point number to a character string. The *value* argument 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 *dec* argument 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 *sign* argument 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.

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.

Return Value

The **ecvt** function returns a pointer to the string of digits. There is no error return.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

atof, atoi, atol, fcvt, gcvt

Example

```
/* ECVT.C: This program uses ecvt to convert a floating-point
 * number to a character string.
 */

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

```
void main()
{
   int     decimal, sign;
   char    *buffer;
   int     precision = 10;
   double  source = 3.1415926535;

   buffer = ecvt( source, precision, &decimal, &sign );
   printf( "source: %2.10f   buffer: '%s'  decimal: %d   sign: %d\n",
           source, buffer, decimal, sign );
}
```

Output

```
source: 3.1415926535   buffer: '3141592654'  decimal: 1   sign: 0
```

Description

Draw ellipses.

#include <graph.h>

short _far _ellipse(short *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

short _far _ellipse_w(short *control*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

short _far _ellipse_wxy(short *control*, **struct _wxycoord _far** **pwxy1*, **struct _wxycoord _far** **pwxy2* **);**

control	Fill flag
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle

Remarks

The **_ellipse** functions draw ellipses or circles. The borders are drawn in the current color. In the **_ellipse** function, the center of the ellipse is the center of the bounding rectangle defined by the view-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

In the **_ellipse_w** function, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

In the **_ellipse_wxy** function, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate pairs (*pwxy1*) and (*pwxy2*).

If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

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

The control option given by **_GFILLINTERIOR** is equivalent to a subsequent call to the **_floodfill** function, using the center of the ellipse as the starting point and the current color (set by **_setcolor**) as the boundary color.

Return Value The **_ellipse** functions return a nonzero value if the ellipse is drawn successfully; otherwise, they return 0.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_arc** functions, **_floodfill**, **_grstatus**, **_lineto** functions, **_pie** functions, **_polygon** functions, **_rectangle** functions, **_setcolor**, **_setfillmask**

Example

```
/* ELLIPSE.C: This program draws a simple ellipse. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _ellipse( _GFILLINTERIOR, 80, 50, 240, 150 );

   /* Strike any key to clear screen. */
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Enables interrupts.

#include <dos.h>

void _enable(void);

Remarks The **_enable** routine enables interrupts by executing an 8086 **STI** machine instruction.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_disable**

Description Terminates an OS/2 thread.

#include <process.h> Multithread version of PROCESS.H

void _far _endthread(void);

Description The **_endthread** function terminates a thread created by **_beginthread.**

Because threads terminate automatically, the **_endthread** function is normally not required. It is used to terminate a thread conditionally.

The OS/2 function **DosExit** should not be used to terminate threads created by the **_beginthread** function. If **DosExit** is used, the results are unpredictable.

Return Value None.

Compatibility ☐ ANSI ☐ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_beginthread**

Example See the example for **_beginthread.**

Description

Tests for end-of-file.

#include <io.h>	Required only for function declarations

int eof(int *handle* **);**

handle	Handle referring to open file

Remarks

The **eof** function determines whether the end of the file associated with *handle* has been reached.

Return Value

The **eof** function returns the value 1 if the current position is end-of-file, or 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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

clearerr, feof, ferror, perror

Example

```
/* EOF.C: This program reads data from a file ten bytes at a time
 * until the end of the file is reached or an error is encountered.
 */

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

void main()
{
   int  fh, count, total = 0;
   char buf[10];

   if( (fh = open( "eof.c", O_RDONLY )) == - 1 )
      exit( 1 );
```

```
   /* 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 up actual bytes read */
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   close( fh );
}
```

Output

```
Number of bytes read = 715
```

Description

Load and execute new child processes.

#include <process.h>	Required only for function declarations

int execl(char **cmdname*, **char** **arg0*, ... **char** **argn*, **NULL);**

int execle(char **cmdname*, **char** **arg0*, ... **char** **argn*, **NULL, char** ***envp* **);**

int execlp(char **cmdname*, **char** **arg0*, ... **char** **argn*, **NULL);**

int execlpe(char **cmdname*, **char** **arg0*, ... **char** **argn*, **NULL, char** ***envp* **);**

int execv(char **cmdname*, **char** ***argv* **);**

int execve(char **cmdname*, **char** ***argv*, **char** ***envp* **);**

int execvp(char **cmdname*, **char** ***argv* **);**

int execvpe(char **cmdname*, **char** ***argv*, **char** ***envp* **);**

cmdname	Path name of file to be executed
arg0, ... argn	List of pointers to arguments
argv	Array of pointers to arguments
envp	Array of pointers to environment settings

Remarks

The **exec** functions load and execute new child processes. When the call is successful in DOS, the child process is placed in the memory previously occupied by the calling process. Under OS/2, calling an **exec** function is equivalent to calling the corresponding function with the **P_NOWAITO** argument specified, followed by a call to the **exit** function. Sufficient memory must be available for loading and executing the child process.

All of the **exec** functions use the same operating system function. The letter(s) at the end of the function name determine the specific variation, as shown in the following list:

Letter	Variation
e	An array of pointers to environment arguments is explicitly passed to the child process.
l	Command-line arguments are passed individually to the **exec** function.
p	Uses the PATH environment variable to find the file to be executed.
v	Command-line arguments are passed to the **exec** function as an array of pointers.

The *cmdname* 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 *cmdname* does not have a file-name extension or does not end with a period (.), the **exec** function searches for the named file; if the search is unsuccessful, it tries the same base name, first with the extension .COM, then with the extension .EXE. If *cmdname* has an extension, only that extension is used in the search. If *cmdname* ends with a period, the **exec** calls search for *cmdname* with no extension. The **execlp, execlpe, execvp,** and **execvpe** routines search for *cmdname* (using the same procedures) in the directories specified by the PATH environment variable.

If *cmdname* contains a drive specifier or any slashes (i.e., if it is a relative path name), the **exec** call searches only for the specified file and no path searching is done.

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 (in real mode only). 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; this argument is *argv*[0] of the child process. Usually, this argument is a copy of the *cmdname* argument. (A different value will not produce an error.) Under versions of DOS earlier than 3.0, the passed value of *arg0* is not available for use in the child process. However, under OS/2 and under DOS versions 3.0 and later, *cmdname* 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 *cmdname*. 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 *cmdname*. 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.

A program executed with one of the **exec** family of functions is always loaded into memory as if the "maximum allocation" field in the program's .EXE file header is set to the default value of 0FFFFH. You can use the EXEMOD utility to change the maximum allocation field of a program; however, such a program invoked with one of the **exec** functions may behave differently from a program invoked directly from the operating-system command line or with one of the **spawn** functions.

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.

You must explicitly flush (using **fflush** or **flushall**) or close any stream prior to the **exec** function call.

Signal settings are not preserved in child processes that are created by calls to **exec** routines. The signal settings are reset to the default in the child 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 (OS/2, and 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

Because of differences in DOS versions 2.0 and 2.1, child processes generated by the **exec** family of functions (or by the equivalent **spawn** functions with the **P_OVERLAY** argument) may cause fatal system errors when they exit. If you are running DOS 2.0 or 2.1, you must upgrade to DOS version 3.0 or later to use these functions.

Bound programs cannot use the **exec** family of functions in real mode.

See Also **abort, atexit, exit, _exit, onexit, spawn** functions, **system**

Example

```
/* EXEC.C: 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.
 */

#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
              };

void main( 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( argv[2], argv[2], "execl", "two", NULL );
         break;
      case '2':
         execle( argv[2], argv[2], "execle", "two", NULL, my_env );
         break;
      case '3':
         execlp( argv[2], argv[2], "execlp", "two", NULL );
         break;
      case '4':
         execlpe( argv[2], argv[2], "execlpe", "two", NULL, my_env );
         break;
      case '5':
         execv( argv[2], args );
         break;
      case '6':
         execve( argv[2], args, my_env );
         break;
      case '7':
         execvp( argv[2], args );
         break;
      case '8':
         execvpe( argv[2], args, my_env );
         break;
      default:
         printf( "SYNTAX: EXEC <1-8> <childprogram>\n" );
         exit( 1 );
   }
   printf( "Process was not spawned.\n" );
   printf( "Program 'child' was not found." );
}
```

Description

Terminate the calling process after cleanup (**exit**) or immediately (**_exit**).

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

void exit(int *status* **);**

void _exit(int *status* **);**

status	Exit status

Remarks

The **exit** and **_exit** functions terminate the calling process. The **exit** function first calls, in LIFO (last-in–first-out) order, the functions registered by **atexit** and **onexit**, then flushes all file buffers 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 one exists, after the calling process exits. The *status* value is available to the operating-system batch command ERRORLEVEL.

The behavior of the **exit**, **_exit**, **_cexit**, and **_c_exit** functions is as follows:

Function	Action
exit	Performs complete C library termination procedures, terminates the process, and exits with the supplied status code.
_exit	Performs "quick" C library termination procedures, terminates the process, and exits with the supplied status code.
_cexit	Performs complete C library termination procedures and returns to caller, but does not terminate the process.
_c_exit	Performs "quick" C library termination procedures and returns to caller, but does not terminate the process.

Return Value None.

Compatibility **exit**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_exit

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **abort, atexit, _cexit, exec** functions, **onexit, spawn** functions, **system**

Example

```
/* EXITER.C: This program prompts the user for a yes or no and returns
 * a DOS error code of 1 if the user answers Y or y; otherwise it
 * returns 0. The error code could be tested in a batch file.
 */

#include <conio.h>
#include <stdlib.h>

void main()
{
   char  ch;

   cputs( "Yes or no? " );
   ch = getch();
   cputs( "\r\n" );
   if( toupper( ch ) == 'Y' )
      exit( 1 );
   else
      exit( 0 );
}
```

Description

Calculate the exponential.

#include <math.h>

double exp(double *x* **);**

long double expl(long double *x* **);**

x	Floating-point value

Remarks

The **exp** and **expl** functions return the exponential function of their floating-point arguments (*x*).

The **expl** function is the 80-bit counterpart; it uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

These functions return e^x. The functions return **HUGE_VAL** on overflow and set **errno** to **ERANGE**; on underflow, they return 0 but do not set **errno.**

Compatibility

exp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

expl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

log functions

Example

```
/* EXP.C */
#include <math.h>
#include <stdio.h>
```

```
void main()
{
   double x = 2.302585093, y;

   y = exp( x );
   printf( "exp( %f ) = %f\n", x, y );
}
```

Output

```
exp( 2.302585 ) = 10.000000
```

Description

Changes the size of a memory block.

#include <malloc.h>	Required only for function declarations

void *_expand(void ***memblock*, size_t** *size* **);**

void _based(void) *_bexpand(_segment *seg***, void _based(void) ***memblock***,
size_t** *size* **);**

void _far *_fexpand(void _far ***memblock*, size_t** *size* **);**

void _near *_nexpand(void _near ***memblock*, size_t** *size* **);**

memblock	Pointer to previously allocated memory block
size	New size in bytes
seg	Value of base segment

Remarks

The **_expand** family of functions 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 *memblock* 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 *memblock* argument can also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **_expand**, **malloc**, or **realloc**. If *memblock* points to a freed block, the block remains free after a call to one of the **_expand** functions.

The *seg* argument is the segment address of the **_based** heap.

In large data models (compact-, large-, and huge-model programs), **_expand** maps to **_fexpand**. In small data models (tiny-, small-, and medium-model programs), **expand** maps to **_nexpand**.

The various **_expand** functions change the size of the storage block in the data segments shown in the list below:

Function	**Data Segment**
_expand	Depends on data model of program
_bexpand	Based heap specified by *seg*, or in all based heaps if *seg* is zero
_fexpand	Far heap (outside default data segment)
_nexpand	Near heap (inside default data segment)

Return Value

The **_expand** family of functions returns a **void** pointer to the reallocated memory block. Unlike **realloc**, **_expand** cannot move a block to change its size. This means the *memblock* argument to **_expand** is the same as the return value if there is sufficient memory available to expand the block without moving it.

With the exception of the **_bexpand** function, these functions return **NULL** if there is insufficient memory available to expand the block to the given size without moving it. The **_bexpand** function returns **_NULLOFF** if insufficient memory is available. The item pointed to by *memblock* 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 **void**, use a type cast on the return value.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

calloc functions, **free** functions, **malloc** functions, **_msize** functions, **realloc** functions

Example

```
/* EXPAND.C */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main()
{
   char *bufchar;

   printf( "Allocate a 512 element buffer\n" );
   if( (bufchar = (char *)calloc( 512, sizeof( char ) )) == NULL )
      exit( 1 );
   printf( "Allocated %d bytes at %Fp\n",
         _msize( bufchar ), (void _far *)bufchar );

   if( (bufchar = (char *)_expand( bufchar, 1024 )) == NULL )
      printf( "Can't expand" );
   else
      printf( "Expanded block to %d bytes at %Fp\n",
            _msize( bufchar ), (void _far *)bufchar );

   /* Free memory */
   free( bufchar );
   exit( 0 );
}
```

Output

```
Allocate a 512 element buffer
Allocated 512 bytes at 0067:142A
Expanded block to 1024 bytes at 0067:142A
```

Description

Calculate the absolute value of their floating-point arguments.

#include <math.h>

double fabs(double *x* **);**

long double fabsl(long double *x* **);**

x	Floating-point value

Remarks

The **fabs** and **fabsl** functions calculate the absolute value of their floating-point arguments.

The **fabsl** function is the 80-bit counterpart; it uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

These functions return the absolute value of their arguments. There is no error return.

Compatibility

fabs

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

fabsl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

abs, cabs, labs

Example

```
/* ABS.C: This program computes and displays the absolute values of
 * several numbers.
 */

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

```
void main()
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;

   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);

   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);

   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

Output

```
The absolute value of -4 is 4
The absolute value of -41567 is 41567
The absolute value of -3.141593 is 3.141593
```

Description

Closes a stream (**fclose**) or closes all open streams (**fcloseall**).

#include <stdio.h>

int fclose(FILE ****stream* **);

int fcloseall(void);

stream	Pointer to **FILE** structure

Remarks

The **fclose** function closes *stream*. The **fcloseall** function closes all open streams except **stdin, stdout, stderr** (and in DOS, **stdaux** and **stdprn**). It also closes and deletes any temporary files created by **tmpfile**.

In both functions, 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.

Compatibility

fclose

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

fcloseall

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

close, fdopen, fflush, fopen, freopen

Example

```
/* FOPEN.C: This program opens files named "data" and "data2". It uses
 * fclose to close "data" and fcloseall to close all remaining files.
 */

#include <stdio.h>
```

```
FILE *stream, *stream2;

void main()
{
   int numclosed;

   /* Open for read (will fail if 'data does not exist) */
   if( (stream  = fopen( "data", "r" )) == NULL )
      printf( "The file 'data' was not opened\n" );
   else
      printf( "The file 'data' was opened\n" );

   /* Open for write */
   if( (stream2 = fopen( "data2", "w+" )) == NULL )
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );

   /* Close stream */
   if( fclose( stream ) )
      printf( "The file 'data' was not closed\n" );

   /* All other files are closed: */
   numclosed = fcloseall( );
   printf( "Number of files closed by fcloseall: %u\n", numclosed );
}
```

Output

```
The file 'data' was opened
The file 'data2' was opened
Number of files closed by fcloseall: 1
```

Description Converts a floating-point number to a string.

#include <stdlib.h>	Required only for function declarations

char *fcvt(double *value*, **int** *count*, **int ****dec*, **int ****sign* **);**

value	Number to be converted
count	Number of digits after decimal point
dec	Pointer to stored decimal-point position
sign	Pointer to stored sign indicator

Remarks The **fcvt** function converts a floating-point number to a null-terminated character string. The *value* argument 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 *count* argument specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, 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 *dec* argument points to an integer value; this integer value gives 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.

The **ecvt** and **fcvt** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the results of the previous call.

Return Value The **fcvt** function returns a pointer to the string of digits. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **atof, atoi, atol, ecvt, gcvt**

Example

```
/* FCVT.C: This program converts the constant 3.1415926535 to a string and
 * sets the pointer *buffer to point to that string.
 */
```

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

void main()
{
   int  decimal, sign;
   char *buffer;
   double source = 3.1415926535;

   buffer = fcvt( source, 7, &decimal, &sign );
   printf( "source: %2.10f   buffer: '%s'   decimal: %d   sign: %d\n",
           source, buffer, decimal, sign );
}
```

Output

```
source: 3.1415926535   buffer: '31415927'   decimal: 1   sign: 0
```

Description

Opens a stream using a handle.

#include <stdio.h>

FILE *fdopen(int *handle*, **char** **mode* **);**

handle	Handle referring to open file
mode	Type of access permitted

Remarks

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 2.7, "Input and Output," for an explanation of stream I/O and low-level I/O.) The *mode* character string specifies the type of access requested for the file, as shown below. The following list gives the *mode* string used in the **fopen** and **fdopen** functions and the corresponding *oflag* arguments used in the **open** and **sopen** functions. A complete description of the *mode* string argument is given in the remarks section of the **fopen** function.

Type String	Equivalent Value for open/sopen
"r"	**O_RDONLY**
"w"	**O_WRONLY** (usually **O_WRONLY | O_CREAT | O_TRUNC**)
"a"	**O_WRONLY | O_APPEND** (usually **O_WRONLY | O_CREAT | O_APPEND**)
"r+"	**O_RDWR**
"w+"	**O_RDWR** (usually **O_RDWR | O_CREAT | O_TRUNC**)
"a+"	**O_RDWR | O_APPEND** (usually **O_RDWR | O_APPEND | O_CREAT**)

In addition to the values listed above, one of the following characters can be included in the *mode* string to specify the translation mode for newlines. These characters correspond to the constants used in the **open** and **sopen** functions, as shown below:

Mode	Equivalent Value for open/sopen
t	**O_TEXT**
b	**O_BINARY**

If **t** or **b** is not given in the *mode* string, the translation mode is defined by the default-mode variable **_fmode.**

The **t** option is not part of the ANSI standard for **fopen** and **fpopen**, but is instead a Microsoft extension and should not be used where ANSI portability is desired.

Return Value The **fdopen** function returns a pointer to the open stream. A null pointer value indicates an error.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **dup, dup2, fclose, fcloseall, fopen, freopen, open**

Example

```
/* FDOPEN.C: This program opens a file using low-level I/O, then uses
 * fdopen to switch to stream access. It counts the lines in the file.
 */

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

void main()
{
   FILE *stream;
   int  fh, count = 0;
   char inbuf[128];

   /* Open a file handle. */
   if( (fh = open( "fdopen.c", O_RDONLY )) == -1 )
      exit( 1 );

   /* Change handle access to stream access. */
   if( (stream = fdopen( fh, "r" )) == NULL )
      exit( 1 );

   while( fgets( inbuf, 128, stream ) != NULL )
      count++;
```

```
   /* After fdopen, close with fclose, not close. */
   fclose( stream );

   printf( "Lines in file: %d\n", count );
}
```

Output

```
Lines in file: 31
```

Description

Tests for end-of-file on a stream.

#include <stdio.h>

int feof(FILE ***stream*** **);**

stream	Pointer to **FILE** structure

Remarks

The **feof** routine (implemented as a macro) determines whether the end of *stream* has been reached. Once the end of the file is reached, read operations return an end-of-file indicator until the stream is closed or until **rewind, fsetpos, fseek,** or **clearerr** 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

clearerr, eof, ferror, perror

Example

```
/* FEOF.C: This program uses feof to indicate when it reaches the end
 * of the file FEOF.C. It also checks for errors with ferror.
 */

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

void main()
{
   int  count, total = 0;
   char buffer[100];
   FILE *stream;

   if( (stream = fopen( "feof.c", "r" )) == NULL )
      exit( 1 );
```

```
   /* Cycle until end of file reached: */
   while( !feof( stream ) )
   {
      /* Attempt to read in 10 bytes: */
      count = fread( buffer, sizeof( char ), 100, stream );
      if( ferror( stream ) )
      {
         perror( "Read error" );
         break;
      }

      /* Total up actual bytes read */
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( stream );
}
```

Output

```
Number of bytes read = 697
```

Description

Tests for an error on a stream.

#include <stdio.h>

int ferror(FILE ****stream* **);**

stream	Pointer to **FILE** structure

Remarks

The **ferror** routine (implemented as a macro) tests for a reading or writing error on the file associated with *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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

clearerr, eof, feof, fopen, perror

Example

```
/* FEOF.C: This program uses feof to indicate when it reaches the end
 * of the file FEOF.C. It also checks for errors with ferror.
 */

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

void main()
{
   int  count, total = 0;
   char buffer[100];
   FILE *stream;

   if( (stream = fopen( "feof.c", "r" )) == NULL )
      exit( 1 );
```

```
   /* Cycle until end of file reached: */
   while( !feof( stream ) )
   {
      /* Attempt to read in 10 bytes: */
      count = fread( buffer, sizeof( char ), 100, stream );
      if( ferror( stream ) )
      {
         perror( "Read error" );
         break;
      }

      /* Total up actual bytes read */
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( stream );
}
```

Output

```
Number of bytes read = 697
```

Description Flushes a stream.

#include <stdio.h>

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

stream Pointer to **FILE** structure

Remarks If the file associated with *stream* is open for output, **fflush** writes to that 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*.

Buffers are automatically flushed when they are full, when the stream is closed, or when a program terminates normally without closing the 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 in which the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fclose, flushall, setbuf**

Example

```
/* FFLUSH.C */
#include <stdio.h>
#include <conio.h>

void main()
{
   int integer;
   char string[81];
```

```
   /* Read each word as a string. */
   printf( "Enter a sentence of four words with scanf: " );
   for( integer = 0; integer < 4; integer++ )
   {
      scanf( "%s", string );
      printf( "%s\n", string );
   }

   /* You must flush the input buffer before using gets. */
   fflush( stdin );
   printf( "Enter the same sentence with gets: " );
   gets( string );
   printf( "%s\n", string );
}
```

Output

```
Enter a sentence of four words with scanf: This is a test
This
is
a
test
Enter the same sentence with gets: This is a test
This is a test
```

Description Read a character from a stream (**fgetc**) or **stdin** (**fgetchar**).

#include <stdio.h>

int fgetc(FILE **stream*);

int fgetchar(void);

stream	Pointer to **FILE** structure

Remarks The **fgetc** function reads a single character from the current position of the file associated with *stream*. The character is converted and returned as an **int**. The function then increments the associated file pointer (if any) to point to the next character. The **fgetchar** function is equivalent to **fgetc(stdin)**.

The **fgetc** and **fgetchar** routines are identical to **getc** and **getchar**, but they are functions rather than macros.

Return Value The **fgetc** and **fgetchar** functions return the character read. They return **EOF** to indicate an error or end-of-file. Use **feof** or **ferror** to distinguish between an error and an end-of-file condition.

Compatibility **fgetc**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

fgetchar

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **fputc, fputchar, getc, getchar**

Example

```
/* FGETC.C: 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.
 */

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

```
void main()
{
   FILE *stream;
   char buffer[81];
   int  i, ch;

   /* Open file to read line from: */
   if( (stream = fopen( "fgetc.c", "r" )) == NULL )
      exit( 0 );

   /* Read in first 80 characters and place them in "buffer": */
   ch = fgetc( stream );
   for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
   {
      buffer[i] = ch;
      ch = fgetc( stream );
   }
   /* Add null to end string */
   buffer[i] = '\0';
   printf( "%s\n", buffer );
   fclose( stream );
}
```

Output

```
/* FGETC.C: This program uses getc to read the first 80 input characters
/* (or
```

Description

Gets a stream's file-position indicator.

#include <stdio.h>

int fgetpos(FILE *_stream_, fpos_t *_pos_);

stream	Target stream
pos	Position-indicator storage

Remarks

The **fgetpos** function gets the current value of the *stream* argument's file-position indicator and stores it in the object pointed to by *pos*. The **fsetpos** function can later use information stored in *pos* to reset the *stream* argument's pointer to its position at the time **fgetpos** was called.

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
EBADF	The specified stream is not a valid file handle or is not accessible.
EINVAL	The *stream* value is invalid.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

fsetpos

Example

```
/* FGETPOS.C: This program opens a file and reads bytes at several
 * different locations.
 */

#include <stdio.h>
```

```
void main()
{
   FILE   *stream;
   fpos_t pos;
   int    val;
   char   buffer[20];

   if( (stream = fopen( "fgetpos.c", "rb" )) == NULL )
      printf( "Trouble opening file\n" );
   else
   {
      /* Read some data and then check the position. */
      fread( buffer, sizeof( char ), 10, stream );
      if( fgetpos( stream, &pos ) != 0 )
         perror( "fgetpos error" );
      else
      {
         fread( buffer, sizeof( char ), 10, stream );
         printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );
      }

      /* Set a new position and read more data */
      pos = 140;
      if( fsetpos( stream, &pos ) != 0 )
         perror( "fsetpos error" );

      fread( buffer, sizeof( char ), 10, stream );
         printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );

      fclose( stream );
   }
}
```

Output

```
10 bytes at byte 10: .C: This p
10 bytes at byte 140:   FILE   *
```

Description

Gets a string from a stream.

#include <stdio.h>

char *fgets(char ***_string_, int** _n_**, FILE ***_stream_ **);**

string	Storage location for data
_n_	Number of characters stored
stream	Pointer to **FILE** structure

Remarks

The **fgets** function reads a string from the input _stream_ argument and stores it in _string_. Characters are read from the current stream position up to and including the first newline 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 newline character, 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 newline 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fputs, gets, puts

Example

```
/* FGETS.C: This program uses fgets to display a line from a file on the
 * screen.
 */

#include <stdio.h>

FILE *stream;

void main()
{
   char line[100], *result;
```

```
   if( (stream = fopen( "fgets.c", "r" )) != NULL )
   {
      if( fgets( line, 100, stream ) == NULL)
         printf( "fgets error\n" );
      else
         printf( "%s", line);
      fclose( stream );
   }
}
```

Output

```
/* FGETS.C: This program uses fgets to display a line from a file on the
```

Description Convert floating-point numbers between IEEE and Microsoft binary formats.

#include <math.h>

int fieeetomsbin(float **src4*, **float** **dst4* **);**

int fmsbintoieee(float **src4*, **float** **dst4* **);**

scr4	Value to be converted
dst4	Converted value

Remarks The **fieeetomsbin** routine converts a single-precision floating-point number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft (MS) 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 stores 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*.

These routines do not handle IEEE NANs ("not a number") and infinities. IEEE denormals are treated as 0 in the conversions.

Return Value These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **dieeetomsbin, dmsbintoieee**

Description Gets the length of a file.

#include <io.h> Required only for function declarations

long filelength(**int** *handle* **);**

handle Target file handle

Remarks The **filelength** function returns the length, in bytes, of the target file associated with *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**.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **chsize, fileno, fstat, stat**

Example

```
/* CHSIZE.C: This program uses filelength to report the size of a
 * file before and after modifying it with chsize.
 */

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

void main()
{
   int fh, result;
   unsigned int nbytes = BUFSIZ;
```

```
   /* Open a file */
   if( (fh = open( "data", O_RDWR | O_CREAT, S_IREAD | S_IWRITE )) != -1 )
   {
      printf( "File length before: %ld\n", filelength( fh ) );
      if( chsize( fh, 329678 ) == 0 )
         printf( "Size successfully changed\n" );
      else
         printf( "Problem in changing the size\n" );
      printf( "File length after:  %ld\n", filelength( fh ) );
      close( fh );
   }
}
```

Output

```
File length before: 0
Size successfully changed
File length after:  329678
```

Description

Gets the file handle associated with a stream.

#include <stdio.h>

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

stream	Pointer to **FILE** structure

Remarks

The **fileno** routine returns the file handle currently associated with *stream*. This routine is implemented as a macro.

Return Value

The **fileno** routine returns the file handle. There is no error return. The result is undefined if *stream* does not specify an open file.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fdopen, filelength, fopen, freopen

Example

```
/* FILENO.C: This program uses fileno to obtain the file handle for
 * some standard C streams.
 */

#include <stdio.h>

void main()
{
   printf( "The file handle for stdin is %d\n", fileno( stdin ) );
   printf( "The file handle for stdout is %d\n", fileno( stdout ) );
   printf( "The file handle for stderr is %d\n", fileno( stderr ) );
}
```

Output

```
The file handle for stdin is 0
The file handle for stdout is 1
The file handle for stderr is 2
```

Description Fill an area of a display using the current color and fill mask

#include <graph.h>

short _far _floodfill(short *x*, **short** *y*, **short** *boundary* **);**

short _far _floodfill_w(double *wx*, **double** *wy*, **short** *boundary* **);**

x, y	Start point
wx, wy	Start point
boundary	Boundary color of area to be filled

Remarks The functions in the **_floodfill** family fill an area of the display, using the current color and fill mask. The **_floodfill** routine begins filling at the view-coordinate point (*x*, *y*). The **_floodfill_w** routine begins filling at the window-coordinate point (*wx*, *wy*).

If this point lies inside the figure, the interior is filled; if it lies 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** functions return 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.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_ellipse** functions, **_getcolor**, **_getfillmask**, **_grstatus**, **_pie** functions, **_setfillmask**, **_setcliprgn**, **_setcolor**

Example

```
/* FLOODFIL.C: This program draws a series of nested rectangles in
 * different colors, constantly changing the background color.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>
```

```
void main()
{
   int loop;
   int xvar, yvar;

   /* find a valid graphics mode */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );

   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() % 16 );
      _floodfill( 0, 0, loop % 16 );
   }
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Calculate the floor of a value.

#include <math.h>

double floor(double *x* **);**

long double floorl(long double *x* **);**

x	Floating-point value

Remarks The **floor** and **floorl** functions return a floating-point value representing the largest integer that is less than or equal to *x*.

The **floorl** function is the 80-bit counterpart, and it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value These functions return the floating-point result. There is no error return.

Compatibility **floor**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

floorl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **ceil, fmod**

Example

```
/* FLOOR.C: This example displays the largest integers less than or equal
 * to the floating-point values 2.8 and -2.8. It then shows the smallest
 * integers greater than or equal to 2.8 and -2.8.
 */

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

```
void main()
{
   double y;

   y = floor( 2.8 );
   printf( "The floor of 2.8 is %f\n", y );
   y = floor( -2.8 );
   printf( "The floor of -2.8 is %f\n", y );

   y = ceil( 2.8 );
   printf( "The ceil of 2.8 is %f\n", y );
   y = ceil( -2.8 );
   printf( "The ceil of -2.8 is %f\n", y );
}
```

Output

```
The floor of 2.8 is 2.000000
The floor of -2.8 is -3.000000
The ceil of 2.8 is 3.000000
The ceil of -2.8 is -2.000000
```

Description Flushes all streams; clears all buffers.

#include <stdio.h>

int flushall(void);

Remarks The **flushall** function writes to its associated files the contents of all buffers associated with open output streams. 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.

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **fflush**

Example

```
/* FLUSHALL.C: This program uses flushall to flush all open buffers. */

#include <stdio.h>

void main()
{
   int numflushed;

   numflushed = flushall();
   printf( "There were %d streams flushed\n", numflushed );
}
```

Output

```
There were 3 streams flushed
```

Description Calculates the floating-point remainder.

#include <math.h>

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

long double fmodl(long double *x*, **long double** *y* **);**

x, *y*	Floating-point values

Remarks The **fmod** and **fmodl** functions calculate the floating-point remainder *f* of *x* / *y* such that $x = i * y + f$, 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*.

The **fmodl** function is the 80-bit counterpart; it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the discussion of the long double functions for more details on this data type.

Return Value These functions return the floating-point remainder. If *y* is 0, the function returns 0.

Compatibility **fmod**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

fmodl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **ceil, fabs, floor**

Example

```
/* FMOD.C: This program displays a floating-point remainder. */

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

```
void main()
{
   double x = -10.0, y = 3.0, z;

   z = fmod( x, y );
   printf( "The remainder of %.2f / %.2f is %f\n", x, y, z );
}
```

Output

```
The remainder of -10.00 / 3.00 is -1.000000
```

Description

Opens a file.

#include <stdio.h>

FILE *fopen(const char **filename*, **const char** **mode* **);**

filename	Path name of file
mode	Type of access permitted

Remarks

The **fopen** function opens the file specified by *filename*. The character string *mode* specifies the type of access requested for the file, as follows:

Type	Description
"r"	Opens for reading. If the file does not exist or cannot be found, the **fopen** 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.

When a file is opened with the **"a"** or **"a+"** access 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+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch 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 included in *mode* to specify the translation mode for newline characters:

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 for 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.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable **_fmode**. If **t** or **b** is prefixed to the argument, the function will fail and return **NULL**.

See Section 2.7, "Input and Output," for a discussion of text and binary modes.

Return Value

The **fopen** function returns a pointer to the open file. A null pointer value indicates an error.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

Note that the **t** option is not part of the ANSI standard for **fopen**; it is a Microsoft extension and should not be used where ANSI portability is desired.

See Also

fclose, fcloseall, fdopen, ferror, fileno, freopen, open, setmode

Example

```
/* FOPEN.C: This program opens files named "data" and "data2". It uses
 * fclose to close "data" and fcloseall to close all remaining files.
 */

#include <stdio.h>
```

```
FILE *stream, *stream2;

void main()
{
   int numclosed;

   /* Open for read (will fail if 'data' does not exist) */
   if( (stream  = fopen( "data", "r" )) == NULL )
      printf( "The file 'data' was not opened\n" );
   else
      printf( "The file 'data' was opened\n" );

   /* Open for write */
   if( (stream2 = fopen( "data2", "w+" )) == NULL )
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );

   /* Close stream */
   if( fclose( stream ) )
      printf( "The file 'data' was not closed\n" );

   /* All other files are closed: */
   numclosed = fcloseall( );
   printf( "Number of files closed by fcloseall: %u\n", numclosed );
}
```

Output

```
The file 'data' was opened
The file 'data2' was opened
Number of files closed by fcloseall: 1
```

Description

Get or set a far-pointer offset (**FP_OFF**) or a far-pointer segment (**FP_SEG**).

#include <dos.h>

unsigned FP_OFF(void _far ***address);**

unsigned FP_SEG(void _far ***address);**

address — Far pointer to memory address

Remarks

The **FP_OFF** and **FP_SEG** macros can be used to set or get the offset and segment, respectively, of the far pointer at *address*.

Return Value

The **FP_OFF** macro returns an offset. The **FP_SEG** macro returns a segment address.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

Example

```
/* FP_SEG.C: This program uses FP_SEG and FP_OFF to obtain
 * the segment and offset of the long pointer p.
 */

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


void main()
{
   void _far *p;
   unsigned int seg_val;
   unsigned int off_val;

   p = _fmalloc( 100 );        /* Points pointer at something */

   seg_val = FP_SEG( p );      /* Gets address pointed to */
   off_val = FP_OFF( p );

   printf( "Segment is %.4X; Offset is %.4X\n", seg_val, off_val );
}
```

Output

```
Segment is 00C7; Offset is 0016
```

Description Resets the floating-point package.

#include <float.h>

void _fpreset(void);

Remarks The **_fpreset** function reinitializes the floating-point-math package. This function is usually used in conjunction with **signal**, **system**, or the **exec** or **spawn** functions.

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**.

In 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. 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 None.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **exec** functions, **signal**, **spawn** functions

Example

```
/* FPRESET.C: This program uses signal to set up a routine for handling
 * floating-point errors.
 */

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>
```

```
jmp_buf mark;                      /* Address for long jump to jump to */
int  fperr;                        /* Global error number */

void fphandler( int sig, int num ); /* Prototypes */
void fpcheck( void );

void main()
{
    double n1, n2, r;
    int jmpret;

    /* Set up floating point error handler. */
    if( signal( SIGFPE, fphandler ) == SIG_ERR )
    {
        fprintf( stderr, "Couldn't set SIGFPE\n" );
        abort();
    }

    /* Save stack environment for return in case of error. First time
     * through, jmpret is 0, so true conditional is executed. If an
     * error occurs, jmpret will be set to -1 and false conditional
     * will be executed.
     */
    jmpret = setjmp( mark );
    if( jmpret == 0 )
    {
        printf( "Test for invalid operation - " );
        printf( "enter two numbers: " );
        scanf( "%lf %lf", &n1, &n2 );

        r = n1 / n2;
        /* This won't be reached if error occurs. */
        printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );

        r = n1 * n2;
        /* This won't be reached if error occurs. */
        printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
    }
    else
        fpcheck();
}
```

```
/* Handles SIGFPE (floating point error) interrupt. */
void fphandler( int sig, int num )
{
    /* Set global for outside check since we don't want to do I/O in the
     * handler.
     */
    fperr = num;

    /* Initialize floating-point package. */
    _fpreset();

    /* Restore calling environment and jump back to setjmp. Return -1
     * so that setjmp will return false for conditional test.
     */
    longjmp( mark, -1 );
}

void fpcheck()
{
    char fpstr[30];

    switch( fperr )
    {
        case FPE_INVALID:
            strcpy( fpstr, "Invalid number" );
            break;

        case FPE_OVERFLOW:
            strcpy( fpstr, "Overflow" );
            break;

        case FPE_UNDERFLOW:
            strcpy( fpstr, "Underflow" );
            break;

        case FPE_ZERODIVIDE:
            strcpy( fpstr, "Divide by zero" );
            break;

        default:
            strcpy( fpstr, "Other floating point error" );
            break;
    }
    printf( "Error %d: %s\n", fperr, fpstr );
}
```

Output

```
Test for invalid operation - enter two numbers: 5 0
Error 131: Divide by zero
```

Description

Prints formatted data to a stream.

#include <stdio.h>

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

stream	Pointer to **FILE** structure
format	Format-control string
argument	Optional arguments

Remarks

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* argument has the same form and function that it does for the **printf** function; see the Remarks section for the **printf** function for more information on *format* and *argument*.

Return Value

The **fprintf** function returns the number of characters printed, or a negative value in the case of an output error.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

cprintf, fscanf, printf, sprintf

Example

```
/* FPRINTF.C: This program uses fprintf to format various data and
 * print them to the file named FPRINTF.OUT. It then displays
 * FPRINTF.OUT on the screen using the system function to invoke
 * the DOS TYPE command.
 */

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

```
FILE *stream;

void main()
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';

   stream = fopen( "fprintf.out", "w" );
   fprintf( stream, "%s%c", s, c );
   fprintf( stream, "%d\n", i );
   fprintf( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf.out" );
}
```

Output

```
this is a string
10
1.500000
```

Description Write a character to a stream (**fputc**) or to **stdout** (**fputchar**).

#include <stdio.h>

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

int fputchar(int *c* **);**

c	Character to be written
stream	Pointer to **FILE** structure

Remarks 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)**.

The **fputc** and **fputchar** routines are similar to **putc** and **putchar**, but are functions rather than macros.

Return Value The **fputc** and **fputchar** functions return the character written. A return value of **EOF** indicates an error.

Compatibility **fputc**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

fputchar

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **fgetc, fgetchar, putc, putchar**

Example

```
/* FPUTC.C: This program uses fputc and fputchar to send a character
 * array to stdout.
 */

#include <stdio.h>
```

```
void main()
{
   char strptr1[] = "This is a test of fputc!!\n";
   char strptr2[] = "This is a test of fputchar!!\n";
   char *p;

   /* Print line to stream using fputc. */
   p = strptr1;
   while( (*p != '\0') && fputc( *(p++), stdout ) != EOF )
      ;

   /* Print line to stream using fputchar. */
   p = strptr2;
   while( (*p != '\0') && fputchar( *(p++) ) != EOF )
      ;
}
```

Output

```
This is a test of fputc!!
This is a test of fputchar!!
```

Description Writes a string to a stream.

#include <stdio.h>

int fputs(const char ****string*, **FILE** ****stream***);**

string	String to be output
stream	Pointer to **FILE** structure

Remarks 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 nonnegative value if it is successful. If an error occurs, it returns **EOF**.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fgets, gets, puts**

Example

```
/* FPUTS.C: This program uses fputs to write a single line to the
 * stdout stream.
 */

#include <stdio.h>

void main()
{
   fputs( "Hello world from fputs.\n", stdout );
}
```

Output

```
Hello world from fputs.
```

Description

Reads data from a stream.

#include <stdio.h>

size_t fread(void **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* **);**

buffer	Storage location for data
size	Item size in bytes
count	Maximum number of items to be read
stream	Pointer to **FILE** structure

Remarks

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 pairs are replaced with single line-feed characters. The replacement has no effect on the file pointer or the return value.

The file-pointer position is indeterminate if an error occurs. The value of a partially read item cannot 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fwrite, read

Example

```
/* FREAD.C: This program opens a file named FREAD.OUT and writes 25
 * characters to the file. It then tries to open FREAD.OUT and
 * read in 25 characters. If the attempt succeeds, the program
 * displays the number of actual items read.
 */
```

```
#include <stdio.h>

void main()
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;

   /* Open file in text mode: */
   if( (stream = fopen( "fread.out", "w+t" )) != NULL )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = 'z' - i;
      /* Write 25 characters to stream */
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );
   }
   else
      printf( "Problem opening the file\n" );

   if( (stream = fopen( "fread.out", "r+t" )) != NULL )
   {
      /* Attempt to read in 25 characters */
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else
      printf( "Was not able to open the file\n" );
}
```

Output

```
Wrote 25 items
Number of items read = 25
Contents of buffer = zyxwvutsrqponmlkjihgfedcb
```

Description

Deallocate a memory block.

#include <stdlib.h>	For ANSI compatibility (**free** only)
#include <malloc.h>	Required only for function declarations

void free(void **memblock* **);**

void _bfree(_segment *seg*, **void _based(void)** **memblock* **);**

void _ffree(void _far **memblock* **);**

void _nfree(void _near **memblock* **);**

memblock	Allocated memory block
seg	Based-heap segment selector

Remarks

The **free** family of functions deallocates a memory block. The argument *memblock* 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.

The *seg* argument specifies the based heap containing the memory block to be freed by the **_bfree** function.

Attempting to free an invalid pointer may affect subsequent allocation and cause errors. An invalid pointer is one not allocated with the appropriate call.

The following restrictions apply to use of the **free**, **_bfree**, **_ffree**, and **_nfree** functions:

Blocks allocated with:	Should be freed with:
calloc, malloc, realloc	**free**
_bcalloc, _bmalloc, _brealloc	**_bfree**
_fcalloc, _fmalloc, _frealloc	**_ffree**
_ncalloc, _nmalloc, _nrealloc	**_nfree**

A **NULL** pointer argument is ignored.

In large data models (compact-, large-, and huge-model programs), **free** maps to **_ffree**. In small data models (tiny-, small-, and medium-model programs), **free** maps to **_nfree**.

The various **free** functions deallocate a memory block in the segments shown in the list below:

Function	**Data Segment**
free	Depends on data model of program
_bfree	Based heap specified by *seg* value
_ffree	Far heap (outside default data segment)
_nfree	Near heap (inside default data segment)

Return Value None.

Compatibility **free**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_bfree, _ffree, _nfree

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **calloc** functions, **malloc** functions, **realloc** functions

Example

```
/* MALLOC.C: This program allocates memory with malloc, then frees
 * the memory with free.
 */

#include <stdlib.h>         /* Definition of _MAX_PATH */
#include <stdio.h>
#include <malloc.h>
```

```
void main()
{
   char *string;

   /* Allocate space for a path name */
   string = malloc( _MAX_PATH );
   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory space allocated for path name\n" );
   free( string );
   printf( "Memory freed\n" );
}
```

Output

```
Memory space allocated for path name
Memory freed
```

Description Returns the amount of memory available for memory allocation.

#include <malloc.h> Required only for function declarations

unsigned int _freect(size_t *size* **);**

size Item size in bytes

Remarks The **_freect** function tells you how much memory is available for dynamic memory allocation in the near heap. It does so by returning the approximate number of times your program can call **_nmalloc** (or **malloc** in small data models) to allocate an item *size* bytes long in the near heap (default data segment).

Return Value The **_freect** function returns the number of calls as an unsigned integer.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **calloc** functions, **_expand** functions, **malloc** functions, **_memavl**, **_msize** functions, **realloc** functions

Example

```
/* FREECT.C: This program determines how much free space is available for
 * integers in the default data segment. Then it allocates space for
 * 1,000 integers and checks the space again, using _freect.
 */

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

void main()
{
   int i;

   /* First report on the free space: */
   printf( "Integers (approximate) available on heap: %u\n\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( "After allocating space for 1000 integers:\n" );
   printf( "Integers (approximate) available on heap: %u\n\n",
           _freect( sizeof( int ) ) );

}
```

Output

```
Integers (approximate) available on heap: 15212

After allocating space for 1000 integers:
Integers (approximate) available on heap: 14084
```

Description

Reassigns a file pointer.

#include <stdio.h>

FILE *freopen(const char ***filename*****, const char** ***mode*****, FILE** ***stream*** **);**

filename	Path name of new file
mode	Type of access permitted
stream	Pointer to **FILE** structure

Remarks

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *filename*. The **freopen** function is typically used to redirect the pre-opened files **stdin**, **stdout**, and **stderr** to files specified by the user. The new file associated with *stream* is opened with *mode*, which is a character string specifying the type of access requested for the file, as follows:

Type	Description
"r"	Opens for reading. If the file does not exist or cannot be found, the **freopen** call fails.
"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 does not 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 does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access 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+" access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch 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 included in the *mode* string to specify the translation mode for newlines.

Mode	Meaning
t	Open in text (translated) mode; carriage-return–line-feed (CR-LF) combinations are translated into single line-feed (LF) characters on input; 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 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.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in the *mode* string, the translation mode is defined by the default mode variable **_fmode**.

See Section 2.7, "Input and Output," for a discussion of text and binary modes.

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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

The **t** option is not part of the ANSI standard for **freopen**; it is a Microsoft extension that should not be used where ANSI portability is desired.

See Also

fclose, fcloseall, fdopen, fileno, fopen, open, setmode

Example

```
/* FREOPEN.C: This program reassigns stdaux to the file
 * named FREOPEN.OUT and writes a line to that file.
 */
```

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

FILE *stream;

void main()
{

   /* Reassign "stdaux" to "freopen.out": */
   stream = freopen( "freopen.out", "w", stdaux );

   if( stream == NULL )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fprintf( stdout, "successfully reassigned\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

Output

```
successfully reassigned
This will go to the file 'freopen.out'
```

Description

Get the mantissa and exponent of a floating-point number.

#include <math.h>

double frexp(double *x*, **int** **expptr* **);**

long double frexpl(long double *x*, **int** **expptr* **);**

x	Floating-point value
expptr	Pointer to stored integer exponent

Remarks

The **frexp** and **frexpl** functions break 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*.

The **frexpl** function is the 80-bit counterpart and uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

These functions return the mantissa. If x is 0, the function returns 0 for both the mantissa and the exponent. There is no error return.

Compatibility

frexp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

frexpl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

ldexp functions, **modf**

Example

```
/* FREXP.C: This program calculates frexp( 16.4, &n ), then displays y
 * and n.
 */

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

```
void main()
{
   double x, y;
   int n;

   x = 16.4;
   y = frexp( x, &n );
   printf( "frexp( %f, &n ) = %f, n = %d\n", x, y, n );
}
```

Output

```
frexp( 16.400000, &n ) = 0.512500, n = 5
```

Description Reads formatted data from a stream.

#include <stdio.h>

int fscanf(FILE ***stream*****, const char** ***format*** **[[,** ***argument*** **]]...)**

stream	Pointer to **FILE** structure
format	Format-control string
argument	Optional arguments

Remarks The **fscanf** function reads data from the current position of *stream* into the locations given by *argument* (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 **scanf** 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 error or end-of-file on *stream* before the first conversion. A return value of 0 means that no fields were assigned.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **cscanf, fprintf, scanf, sscanf**

Example

```
/* FSCANF.C: This program writes formatted data to a file. It
 * then uses fscanf to read the various data back from the file.
 */

#include <stdio.h>
```

```
FILE *stream;

void main()
{
   long l;
   float fp;
   char s[81];
   char c;
   int result;

   stream = fopen( "fscanf.out", "w+" );
   if( stream == NULL )
      printf( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf( stream, "%s %ld %f%c", "a-string", 65000, 3.14159, 'x' );

      /* Set pointer to beginning of file: */
      fseek( stream, 0L, SEEK_SET );

      /* Read data back from file: */
      fscanf( stream, "%s", s );
      fscanf( stream, "%ld", &l );
      fscanf( stream, "%f", &fp );
      fscanf( stream, "%c", &c );

      /* Output data read: */
      printf( "%s\n", s );
      printf( "%ld\n", l );
      printf( "%f\n", fp );
      printf( "%c\n", c );

      fclose( stream );
   }
}
```

Output

```
a-string
65000
3.141590
x
```

Description

Moves the file pointer to a specified location.

#include <stdio.h>

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

stream	Pointer to **FILE** structure
offset	Number of bytes from *origin*
origin	Initial position

Remarks

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_CUR	Current position of file pointer
SEEK_END	End of file
SEEK_SET	Beginning 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*.

When a file is opened for appending data, the current file position is determined by the last I/O operation, not by 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **ftell, lseek, rewind**

Example

```
/* FSEEK.C: This program opens the file FSEEK.OUT and
 * moves the pointer to the file's beginning.
 */

#include <stdio.h>

void main()
{
   FILE *stream;
   char line[81];
   int  result;

   stream = fopen( "fseek.out", "w+" );
   if( stream == NULL )
      printf( "The file fseek.out was not opened\n" );
   else
   {
      fprintf( stream, "The fseek begins here: "
                       "This is the file 'fseek.out'.\n" );
      result = fseek( stream, 23L, SEEK_SET);
      if( result )
         perror( "Fseek failed" );
      else
      {
         printf( "File pointer is set to middle of first line.\n" );
         fgets( line, 80, stream );
         printf( "%s", line );
      }
      fclose( stream );
   }
}
```

Output

```
File pointer is set to middle of first line.
This is the file 'fseek.out'.
```

Description

Sets the stream-position indicator.

#include <stdio.h>

int fsetpos(FILE ***stream*****, const fpos_t** ***pos*** **) ;**

stream	Target stream
pos	Position-indicator storage

Remarks

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 ERRNO.H):

Constant	Meaning
EBADF	The object that *stream* points to is not a valid file handle, or the file is not accessible.
EINVAL	An invalid *stream* value was passed.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

fgetpos

Example

```
/* FGETPOS.C: This program opens a file and reads bytes at several
 * different locations.
 */

#include <stdio.h>
```

```
void main()
{
   FILE   *stream;
   fpos_t pos;
   int    val;
   char   buffer[20];

   if( (stream = fopen( "fgetpos.c", "rb" )) == NULL )
      printf( "Trouble opening file\n" );
   else
   {
      /* Read some data and then check the position. */
      fread( buffer, sizeof( char ), 10, stream );
      if( fgetpos( stream, &pos ) != 0 )
         perror( "fgetpos error" );
      else
      {
         fread( buffer, sizeof( char ), 10, stream );
         printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );
      }

      /* Set a new position and read more data. */
      pos = 140;
      if( fsetpos( stream, &pos ) != 0 )
         perror( "fsetpos error" );

      fread( buffer, sizeof( char ), 10, stream );
         printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );

      fclose( stream );
   }
}
```

Output

```
10 bytes at byte 10: .C: This p
10 bytes at byte 140:   FILE   *
```

Description

Opens a stream with file sharing.

#include <stdio.h>

#include <share.h> *shflag* constants

FILE *_fsopen(const char **filename*, **const char** **mode*, **int** *shflag* **);**

filename	File name to open
mode	Type of access permitted
shflag	Type of sharing allowed

Remarks

The **_fsopen** function opens the file specified by *filename* as a stream and prepares the file for subsequent shared reading or writing, as defined by the *mode* and *shflag* arguments.

The character string *mode* specifies the type of access requested for the file, as follows:

Type	Description
"r"	Opens for reading. If the file does not exist or cannot be found, the **_fsopen** 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 does not 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 does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access 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+" access 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 included in *mode* to specify the translation mode for newlines:

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, **_fsopen** 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.
b	Open in binary (untranslated) mode; the above translations are suppressed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable **_fmode.** If **t** or **b** is prefixed to the argument, the function will fail and will return **NULL.**

See Section 2.7, "Input and Output," for a discussion of text and binary modes.

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 DOS— is not installed, DOS ignores the sharing mode. (See your system documentation for detailed information about sharing modes.)

Constant	Meaning
SH_COMPAT	Sets compatibility mode (not available in OS/2)
SH_DENYNO	Permits read and write access
SH_DENYRD	Denies read access to file
SH_DENYRW	Denies read and write access to file
SH_DENYWR	Denies write access to file

The **_fsopen** function should be used only under OS/2 and DOS versions 3.0 and later. Under earlier versions of DOS, the *shflag* argument is ignored.

Return Value The **_fsopen** function returns a pointer to the stream. A **NULL** pointer value indicates an error.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **fclose, fcloseall, fdopen, ferror, fileno, fopen, freopen, open, setmode, sopen**

Example

```
/* FSOPEN.C: This program opens files named "data" and "data2". It uses
 * fclose to close "data" and fcloseall to close all remaining files.
 */

#include <stdio.h>
#include <share.h>

FILE *stream;

void main()
{
   FILE *stream;

   /* Open output file for writing. Using _fsopen allows us to ensure
    * that no one else writes to the file while we are writing to it.
    */
   if( (stream = _fsopen( "outfile", "wt", SH_DENYWR )) != NULL )
   {
      fprintf( stream, "No one else in the network can write "
                       "to this file until we are done.\n" );
      fclose( stream );
   }
   /* Now others can write to the file while we read it. */
   system( "type outfile" );
}
```

Output

```
No one else in the network can write to this file until we are done.
```

Description

Gets information about an open file.

#include <sys\types.h>

#include <sys\stat.h>

int fstat(int *handle*, struct stat *buffer);

handle	Handle of open file
buffer	Pointer to structure to store results

Remarks

The **fstat** function obtains information about the open file associated with *handle* and stores it in the structure pointed to by *buffer*. 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. The **S_IFCHR** bit is set if *handle* refers to a device. The **S_IFREG** bit is set if *handle* refers to an ordinary file. The read/write bits are set according to the file's permission mode. (**S_IFCHR** and other constants are defined in SYS\ STAT.H.)
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.

If *handle* refers to a device, the size and time fields in the **stat** structure are not meaningful.

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.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

In OS/2, the **st_dev** field does not contain meaningful information. In fact, it is set to zero. OS/2 provides no way to recover the host drive from just the open file handle.

See Also **access, chmod, filelength, stat**

Example

```
/* FSTAT.C: This program uses fstat to report the size of a file
 * named FSTAT.OUT.
 */

#include <io.h>
#include <fcntl.h>
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
   struct stat buf;
   int fh, result;
   char buffer[] = "A line to output";

   if( (fh = open( "fstat.out", O_CREAT | O_WRONLY | O_TRUNC )) == -1 )
      exit( 1 );
   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" );
   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 ) );
   }
   close( fh );
}
```

Output

```
File size     : 16
Drive number  : 0
Time modified : Thu Jun 15 21:38:46 1989
```

Description

Gets the current position of a file pointer.

#include <stdio.h>

long ftell(FILE ***stream* **);**

stream	Target **FILE** structure

Remarks

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 that when a file is opened for appending data, the current file position is determined by the last I/O operation, not by 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. (When a file is opened for appending, the file position is moved to end-of-file before any write operation.) If no I/O operation has yet occurred on a file opened for appending, the file position is the beginning of the file.

Return Value

The **ftell** function returns the current file position. 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 return to file locations correctly. 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fgetpos, fseek, lseek, tell**

Example

```
/* FTELL.C: This program opens a file named FTELL.C for reading and
 * tries to read 100 characters. It then uses ftell to determine the
 * position of the file pointer and displays this position.
 */

#include <stdio.h>

FILE *stream;

void main()
{
   long position;
   char list[100];

   if( (stream = fopen( "ftell.c", "rb" )) != NULL )
   {
      /* Move the pointer by reading data: */
      fread( list, sizeof( char ), 100, stream );

      /* Get position after read: */
      position = ftell( stream );
      printf( "Position after trying to read 100 bytes: %ld\n", position );
      fclose( stream );
   }
}
```

Output

```
Position after trying to read 100 bytes: 100
```

Description

Gets the current time.

#include <sys\types.h>

#include <sys\timeb.h>

void ftime(struct timeb ****timeptr* **);**

timeptr	Pointer to structure defined in SYS\TIMEB.H

Remarks

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 (**dstflag, millitm, time**, and **timezone**), 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. The last digit is always 0 since **millitm** is incremented to the nearest one-hundredth of a second.
time	Time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.
timezone	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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **asctime, ctime, gmtime, localtime, time, tzset**

Example

```
/* FTIME.C: This program uses ftime to obtain the current time
 * and then stores this time in timebuffer.
 */

#include <stdio.h>
#include <sys\timeb.h>
#include <time.h>

void main()
{

   struct timeb timebuffer;
   char *timeline;

   ftime( &timebuffer );
   timeline = ctime( & ( timebuffer.time ) );

   printf( "The time is %.19s.%hu %s",
           timeline, timebuffer.millitm, &timeline[20] );
}
```

Output

```
The time is Thu Jun 15 21:40:34.870 1989
```

Description Makes an absolute path name from a relative path name.

#include <stdlib.h>

char *_fullpath(char ***buffer*****, const char** ***pathname*****, size_t** *maxlen* **);**

buffer	Full path-name buffer
pathname	Relative path name
maxlen	Length of the buffer pointed to by *buffer*

Remarks The **_fullpath** routine converts the partial path stored in *pathname* to a fully qualified path that is stored in *buffer*. Unlike **_makepath**, the **_fullpath** routine can be used with **.** and **..** in the path.

If the length of the fully qualified path is greater than the value of *maxlen*, then **NULL** is returned; otherwise, the address of *buffer* is returned.

If the *buffer* is **NULL**, **_fullpath** will allocate a buffer of **MAX_PATH** size and the *maxlen* argument is ignored.

If the *pathname* argument specifies a disk drive, the current directory of this drive is combined with the path. If the drive is not valid, **_fullpath** returns **NULL**.

Return Value The **_fullpath** function returns a pointer to the buffer containing the absolute path (*buffer*). If there is an error, **_fullpath** returns **NULL**.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **getcwd, _getdcwd, _makepath, _splitpath**

Example

```
/* FULLPATH.C: This program demonstrates how _fullpath creates a full
 * path from a partial path.
 */

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

```
char full[_MAX_PATH], part[_MAX_PATH];

void main()
{
    while( 1 )
    {
        printf( "Enter partial path or ENTER to quit: " );
        gets( part );
        if( part[0] == 0 )
            break;

        if( _fullpath( full, part, _MAX_PATH ) != NULL )
            printf( "Full path is: %s\n", full );
        else
            printf( "Invalid path\n" );
    }
}
```

Output

```
Enter partial path or ENTER to quit: ..
Full path is: C:\
Enter partial path or ENTER to quit: ..\include
Full path is: C:\include
Enter partial path or ENTER to quit: p:
Full path is: P:\
Enter partial path or ENTER to quit: fullpath.c
Full path is: C:\LIBREF\fullpath.c
Enter partial path or ENTER to quit:
```

Description

Writes data to a stream.

#include <stdio.h>

size_t fwrite(const void **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* **);**

buffer	Pointer to data to be written
size	Item size in bytes
count	Maximum number of items to be written
stream	Pointer to **FILE** structure

Remarks

The **fwrite** function writes up to *count* items, of length *size* each, 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fread, write

Example

```
/* FREAD.C: This program opens a file named FREAD.OUT and writes 25
 * characters to the file. It then tries to open FREAD.OUT and
 * read in 25 characters. If the attempt succeeds, the program
 * displays the number of actual items read.
 */

#include <stdio.h>
```

```
void main()
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;

   /* Open file in text mode: */
   if( (stream = fopen( "fread.out", "w+t" )) != NULL )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = 'z' - i;
      /* Write 25 characters to stream */
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );
   }
   else
      printf( "Problem opening the file\n" );

   if( (stream = fopen( "fread.out", "r+t" )) != NULL )
   {
      /* Attempt to read in 25 characters */
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else
      printf( "Was not able to open the file\n" );
}
```

Output

```
Wrote 25 items
Number of items read = 25
Contents of buffer = zyxwvutsrqponmlkjihgfedcb
```

Description

Converts a floating-point value to a string, which it stores in a buffer.

#include <stdlib.h>	Required only for function declarations

char *gcvt(double *value*, **int** *digits*, **char** **buffer* **);**

value	Value to be converted
digits	Number of significant digits stored
buffer	Storage location for result

Remarks

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 appended automatically. There is no provision for overflow.

The **gcvt** function attempts to produce *digits* significant digits in decimal format. If this is not possible, 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.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

atof, atoi, atol, ecvt, fcvt

Example

```
/* GCVT.C: This program converts -3.1415e5 to its string representation. */

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

```
void main()
{
   char buffer[50];
   double source = -3.1415e5;

   gcvt( source, 7, buffer );
   printf( "source: %f  buffer: '%s'\n", source, buffer );
}
```

Output

```
source: -314150.000000  buffer: '-314150.'
```

Description Gets the current active page number.

#include <graph.h>

short _far _getactivepage(void);

Remarks The **_getactivepage** function returns the number of the current active page.

Return Value The function returns the number of the current active page. All hardware combinations support at least one page (page number 0). In OS/2, only page 0 is valid.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getactivepage, _getvideoconfig, _getvisualpage, _grstatus, _setactivepage, _setvideomode, _setvisualpage**

Example

```
/* PAGE.C illustrates video page functions including:
 *      _getactivepage  _getvisualpage  _setactivepage  _setvisualpage
 */

#include <conio.h>
#include <graph.h>
#include <stdlib.h>

void main()
{
   short  oldvpage, oldapage, page, row, col, line;
   struct videoconfig vc;
   char   buf[80];

   _getvideoconfig( &vc );
   if( vc.numvideopages < 4 )
       exit( 1 );              /* Fail for OS/2 or monochrome. */
   oldapage  = _getactivepage();
   oldvpage  = _getvisualpage();
   _displaycursor( _GCURSOROFF );
```

```
    /* Draw arrows in different place on each page. */
    for( page = 1; page < 4; page++ )
    {
       _setactivepage( page );
       _settextposition( 12, 16 * page );
       _outtext( ">>>>>>>>" );
    }

    while( !kbhit() )
       /* Cycle through pages 1 to 3 to show moving image. */
       for( page = 1; page < 4; page++ )
           _setvisualpage( page );
    getch();

    /* Restore original page (normally 0) to restore screen. */
    _setactivepage( oldapage );
    _setvisualpage( oldvpage );
    _displaycursor( _GCURSORON );
}
```

Description

Determines the endpoints in viewport coordinates of the most recently drawn arc or pie.

#include <graph.h>

short _far _getarcinfo(struct xycoord _far **start*, **struct xycoord _far** **end*,
struct xycoord _far **fillpoint* **);**

start	Starting point of arc
end	Ending point of arc
fillpoint	Point at which pie fill will begin

Remarks

The **_getarcinfo** function determines the endpoints in viewport coordinates of the most recently drawn arc or pie.

If successful, the **_getarcinfo** function updates the *start* and *end* **xycoord** structures to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to one of the **_arc** or **_pie** functions.

In addition, *fillpoint* specifies a point from which a pie can be filled. This is useful for filling a pie in a color different from the border color. After a call to **_getarcinfo**, change colors using the **_setcolor** function. Use the color, along with the coordinates in *fillpoint*, as arguments for the *floodfill* function.

Return Value

The **_getarcinfo** function returns a nonzero value if successful. If neither the **_arc** nor the **_pie** function has been successfully called since the last time the screen was cleared or a new graphics mode or viewport was selected, the **_getarcinfo** function returns 0.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_arc functions, **_floodfill**, **_getvideoconfig**, **_grstatus**, **_pie** functions

Example

See the example for **_arc**.

Description

Gets the current background color.

#include <graph.h>

long _far _getbkcolor(void);

Remarks

The **_getbkcolor** function returns the current background color. The default is 0.

In a color text mode such as **_TEXTC80**, **_setbkcolor** accepts, and **_getbkcolor** returns, a color index. For example, **_setbkcolor(2L)** sets the background color to color index 2. The actual color displayed depends on the palette mapping for color index 2. The default for color index 2 is green in a color text mode.

In a color graphics mode such as **_ERESCOLOR**, **_setbkcolor** accepts and **_getbkcolor** returns a color value (as used in **_remappalette**). The value for the simplest background colors is given by the manifest constants defined in the GRAPH.H include file. For example, **_setbkcolor(_GREEN)** sets the background color in a graphics mode to green. These manifest constants are provided as a convenience in defining and manipulating the most common colors. In general, the actual range of colors is much greater.

In most cases, whenever an argument is long, it refers to a color value, and whenever it is short, it refers to a color index. The two exceptions are **_setbkcolor** and **_getbkcolor**, described above. For a more complete discussion of colors, see **_remappalette.**

Return Value

The function returns the current background color value. There is no error return.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

_remappalette, _setbkcolor

Example

See the example for **_getcolor.**

Description

Reads a character from a stream (**getc**), or gets a character from **stdin** (**getchar**).

#include <stdio.h>

int getc(FILE **stream* **);**

int getchar(void);

stream	Current stream

Remarks

The **getc** macro reads a single character from the *stream* position and increments the associated file pointer (if there is one) to point to the next character. The **getchar** macro is identical to **getc(stdin)**.

The **getc** and **getchar** routines are similar to **fgetc** and **fgetchar**, respectively, but are macros rather than functions.

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.

Compatibility

getc

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

getchar

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fgetc, fgetchar, getch, getche, putc, putchar, ungetc

Example

```
/* GETC.C: 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.
 */

#include <stdio.h>
```

```
void main()
{
   char buffer[81];
   int i, ch;

   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 );
}
```

Output

```
Enter a line: This is a line of text.
This is a line of text.
```

Description Get a character from the console without echo (**getch**) or with echo (**getche**).

#include <conio.h> Required only for function declarations

int getch(void);

int getche(void);

Remarks The **getch** function reads a single character from the console without echoing. 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 0xE0, and the second call returns the actual key code.

Return Value The **getch** function returns the character read. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **cgets, getchar, ungetch**

Example

```
/* GETCH.C: This program reads characters from the keyboard until it
 * receives a 'Y' or 'y'.
 */

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

```
void main()
{
   int ch;

   cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = getch();
      ch = toupper( ch );
   } while( ch != 'Y' );

   putch( ch );
   putch( '\r' );    /* Carriage return */
   putch( '\n' );    /* Line feed       */
}
```

Output

```
Type 'Y' when finished typing keys: Y
```

Description Gets the current color.

#include <graph.h>

short _far _getcolor(void);

Remarks The **_getcolor** function returns the current graphics color index. The default is the highest legal value of the current palette.

Return Value The **_getcolor** function returns the current color index.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_setcolor**

Example

```
/* OUTTXT.C: This example illustrates text output functions:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition
 */

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

char buffer [80];

void main()
{

   /* Save original foreground, background, and text position. */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct rccoord oldpos;

   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );
```

```
    /* First time no blink, second time blinking. */
    for( blink = 0; blink <= 16; blink += 16 )
    {
       /* Loop through 8 background colors. */
       for( bgd = 0; bgd < 8; bgd++ )
       {
          _setbkcolor( bgd );
          _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
          _settextcolor( 7 );
          sprintf(buffer, "Back: %d Fore:", bgd );
          _outtext( buffer );

          /* Loop through 16 foreground colors. */
          for( fgd = 0; fgd < 16; fgd++ )
          {
             _settextcolor( fgd + blink );
             sprintf( buffer, " %2d ", fgd + blink );
             _outtext( buffer );
          }
       }
    }
    getch();

    /* Restore original foreground, background, and text position. */
    _settextcolor( oldfgd );
    _setbkcolor( oldbgd );
    _clearscreen( _GCLEARSCREEN );
    _settextposition( oldpos.row, oldpos.col );
}
```

Description

Get the current position and return it as a structure.

#include <graph.h>

struct xycoord _far _getcurrentposition(void);

struct _wxycoord _far _getcurrentposition_w(void);

Remarks

The **_getcurrentposition** functions return the coordinates of the current graphics output position. The **_getcurrentposition** function returns the position as an **xycoord** structure, defined in GRAPH.H.

The **xycoord** structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The **_getcurrentposition_w** function returns the position as an **_wxycoord** structure, defined in GRAPH.H.

The **_wxycoord** structure contains the following elements:

Element	Description
double wx	window *x* coordinate
double wy	window *y* coordinate

The current position can be changed by the **_lineto**, **_moveto**, and **_outgtext** functions.

The default position, set by **_setvideomode**, **_setvideomoderows**, or **_setviewport**, is the center of the viewport.

Only graphics output starts at the current position; these functions do not affect text output, which begins at the current text position. (See **_settextposition** for more information.)

Return Value

The **_getcurrentposition** function returns the coordinates of the current graphics output position. There is no error return.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_grstatus**, **_lineto** functions, **_moveto** functions, **_outgtext**

Example

```
/* GCURPOS.C: This program sets a random current location, then gets that
 * location with _getcurrentposition.
 */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>

char   buffer[255];

void main()
{
   struct videoconfig vc;
   struct xycoord position;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );

   /* Move to random location and report that location. */
   _moveto( rand() % vc.numxpixels, rand() % vc.numypixels );
   position = _getcurrentposition();
   sprintf( buffer, "x = %d, y = %d", position.xcoord, position.ycoord );
   _settextposition( 1, 1 );
   _outtext( buffer );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Gets the current working directory.

#include <direct.h> Required only for function declarations

char *getcwd(char ***_buffer_, int** _maxlen_ **);**

buffer	Storage location for path name
maxlen	Maximum length of path name

Remarks The **getcwd** function gets the full path name of the current working directory and stores it at _buffer_. The integer argument _maxlen_ specifies the maximum length for the path name. An error occurs if the length of the path name (including the terminating null character) exceeds _maxlen_.

The _buffer_ argument can be **NULL**; a buffer of at least size _maxlen_ (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 a pointer to _buffer_. A **NULL** return value indicates an error, and **errno** is set to one of the following values:

Value	Meaning
ENOMEM	Insufficient memory to allocate _maxlen_ bytes (when a **NULL** argument is given as _buffer_)
ERANGE	Path name longer than _maxlen_ characters

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **chdir, mkdir, rmdir**

Example

```
/* 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 _MAX_DIR leaves room for the longest legal
 * directory name.
 */
```

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

void main()
{
   char buffer[_MAX_DIR];

   /* Get the current working directory: */
   if( getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

Output

```
C:\LIBREF
```

Description

Gets full path name of current working directory, including disk drive.

#include <direct.h>	Required only for function declarations

char *_getdcwd(int *drive*, **char ****buffer*, **int** *maxlen* **);**

drive	Disk drive
buffer	Storage location for path name
maxlen	Maximum length of path name

Remarks

The **_getdcwd** function gets the full path name of the current working directory, including disk drive specification, and stores it at *buffer*. The argument *maxlen* specifies the maximum length for the path name. An error occurs if the length of the path name (including the terminating null character) exceeds *maxlen*.

The *drive* argument specifies the drive (0 = default drive, 1=A, 2=B, etc.). The *buffer* argument can be **NULL**; a buffer of at least size *maxlen* (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 **_getdcwd** return value (a pointer to the allocated buffer).

Return Value

The **_getdcwd** function returns *buffer*. A **NULL** return value indicates an error, and **errno** is set to one of the following values:

Value	Meaning
ENOMEM	Insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*)
ERANGE	Path name longer than *maxlen* characters

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

chdir, getcwd, _getdrive, mkdir, rmdir

Example

```
/* GETDRIVE.C illustrates drive functions including:
 *      _getdrive       _chdrive        _getdcwd
 */
```

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

void main()
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];

   /* Save current drive. */
   curdrive = _getdrive();

   printf( "Available drives are: \n" );

   /* If we can switch to the drive, it exists. */
   for( drive = 1; drive <= 26; drive++ )
      if( !_chdrive( drive ) )
         printf( "%c: ", drive + 'A' - 1 );

   while( 1 )
   {
      printf( "\nType drive letter to check or ESC to quit: " );
      ch = getch();
      if( ch == 27 )
         break;
      if( isalpha( ch ) )
         putch( ch );
      if( _getdcwd( toupper( ch ) - 'A' + 1, path, _MAX_PATH ) != NULL )
         printf( "\nCurrent directory on that drive is %s\n", path );
   }

   /* Restore original drive. This is only necessary for DOS. Under OS/2
    * the current drive of the calling process is always restored.
    */
   _chdrive( curdrive );
   printf( "\n" );
}
```

Output

```
Available drives are:
A: B: C:
Type drive letter to check or ESC to quit: q
Type drive letter to check or ESC to quit: a
Current directory on that drive is A:\

Type drive letter to check or ESC to quit: c
Current directory on that drive is C:\LIBREF

Type drive letter to check or ESC to quit:
```

Description Gets the current disk drive.

#include <direct.h>

int _getdrive(void);

Remarks The **_getdrive** function returns the current working drive (1=A, 2=B, etc.).

Return Value The return value is stated above. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_chdrive, _dos_getdrive, _dos_setdrive, _getcwd, _getdcwd**

Example See the example for **_getdcwd.**

Description

Gets a value from the environment table.

#include <stdlib.h> Required only for function declarations

char *getenv(const char ****varname* **);

varname Name of environment variable

Remarks

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.) Because the **getenv** function is case sensitive, the *varname* variable should match the case of the environment variable.

The **getenv** function returns a pointer to an entry in the environment table. It is, however, only safe to retrieve the value of the environment variable using the returned pointer. To modify the value of an environmental variable, use the **putenv** function.

The **getenv** and **putenv** functions use the copy of the environment contained in the global variable **environ** to access the environment. Programs that use the *envp* argument to **main** and the **putenv** function may retrieve invalid information. The safest programming practice is to use **getenv** and **putenv**.

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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

The **getenv** function operates only on the data structures accessible to the run-time library and not on the environment "segment" created for a process by DOS or OS/2.

See Also

putenv

Example

```
/* GETENV.C: This program uses getenv to retrieve the LIB environment
 * variable and then uses putenv to change it to a new value.
 */

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

```
main()
{
   char *libvar;

   /* Get the value of the LIB environment variable. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );

   /* Attempt to change path. Note that this only affects the environment
    * variable of the current process. The command processor's environment
    * is not changed.
    */
   putenv( "LIB=c:\\mylib;c:\\yourlib" );

   /* Get new value. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );

}
```

Output

```
Original LIB variable is: C:\LIB
New LIB variable is: c:\mylib;c:\yourlib
```

Description Gets the current fill mask for some graphics routines.

#include <graph.h>

unsigned char _far * _far _getfillmask(unsigned char _far ****mask*** **);**

mask	Mask array

Remarks Some graphics routines (**_ellipse**, **_floodfill**, **_pie**, **_polygon**, and **_rectangle**) can fill part or all of the screen with the current color or background color. The fill mask controls the pattern used for filling.

The **_getfillmask** function returns the current fill mask. The mask is an 8-by-8-bit array, in which each bit represents a pixel. If the bit is 1, the corresponding pixel is set to the current color; if the bit is 0, the pixel is left unchanged. The mask is repeated over the entire fill area. If no fill mask is set, or if *mask* is NULL, a solid (unpatterned) fill is performed using the current color.

Return Value If no mask is set, the function returns **NULL**.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_ellipse** functions, **_floodfill**, **_pie** functions, **_polygon** functions, **_rectangle** functions, **_setfillmask**

Example

```
/* GFILLMSK.C: This program illustrates _getfillmask and _setfillmask. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>
```

```
void ellipsemask( short x1, short y1, short x2, short y2, char _far *newmask );

unsigned char mask1[8] = { 0x43, 0x23, 0x7c, 0xf7, 0x8a, 0x4d, 0x78, 0x39 };
unsigned char mask2[8] = { 0x18, 0xad, 0xc0, 0x79, 0xf6, 0xc4, 0xa8, 0x23 };
char oldmask[8];

void main()
{
   int loop;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   /* Set first fill mask and draw rectangle. */
   _setfillmask( mask1 );
   _rectangle( _GFILLINTERIOR, 20, 20, 100, 100 );
   getch();

   /* Call routine that saves and restores mask. */
   ellipsemask( 60, 60, 150, 150, mask2 );
   getch();

   /* Back to original mask. */
   _rectangle( _GFILLINTERIOR, 120, 120, 190, 190 );
   getch();

   _setvideomode( _DEFAULTMODE );
}

/* Draw an ellipse with a specified fill mask. */
void ellipsemask( short x1, short y1, short x2, short y2, char _far *newmask )
{
   unsigned char savemask[8];

   _getfillmask( savemask );                     /* Save mask         */
   _setfillmask( newmask );                      /* Set new mask      */
   _ellipse( _GFILLINTERIOR, x1, y1, x2, y2 );  /* Use new mask      */
   _setfillmask( savemask );                     /* Restore original  */
}
```

Description Gets the current font characteristics.

#include <graph.h>

short _far _getfontinfo(struct _fontinfo _far ***fontbuffer* **);**

fontbuffer	Buffer to hold font information

Remarks The **_getfontinfo** function gets the current font characteristics and stores them in a **_fontinfo** structure, defined in GRAPH.H.

The **_fontinfo** structure contains the following elements:

Element	Contents
int type	Specifies vector (1) or bit-mapped (0) font
int ascent	Specifies pixel distance from top to baseline
int pixwidth	Specifies the character width in pixels; 0 indicates a proportional font
int pixheight	Specifies the character height in pixels
int avgwidth	Specifies the average character width in pixels
char filename [81]	Specifies the file name, including the path
char facename [32]	Specifies the font name

Return Value The **_getfontinfo** function returns a negative number if a font has not been registered or loaded.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getgtextextent, _outgtext, _registerfonts, _setfont, _setgtextvector, _unregisterfonts**

Example See the example for **_outgtext**.

Description Gets the width in pixels of font-based text.

#include <graph.h>

short _far _getgtextextent(unsigned char _far ***text*** **);**

text	Text to be analyzed

Remarks The **_getgtextextent** function returns the width in pixels that would be required to print the *text* string using **_outgtext** with the current font.

This function is particularly useful for determining the size of text that uses proportionally spaced fonts.

Return Value The **_getgtextextent** function returns the width in pixels. It returns –1 if a font has not been registered.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getfontinfo, _outgtext, _registerfonts, _setfont, _unregisterfonts**

Example See the example for **_outgtext.**

Description

Changes the orientation of font text output.

#include <graph.h>

struct xycoord _far _getgtextvector(void);

Remarks

The **_getgtextvector** function gets the current orientation for font text output. The current orientation is used in calls to the **_outgtext** function.

The text-orientation vector, which determines the direction of font-text rotation on the screen, is returned in a structure of type **xycoord**. The *xcoord* and *ycoord* members of the structure describe the vector. The text-rotation options are shown below:

(x, y)	Text Orientation
(1,0)	Horizontal text (default)
(0,1)	Rotated 90 degrees counterclockwise
(-1,0)	Rotated 180 degrees
(0,-1)	Rotated 270 degrees counterclockwise

Return Value

The **_getgtextvector** function returns the current text-orientation vector in a structure of type **xycoord**.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getgtextextent, _grstatus, _outgtext, _setfont, _setgtextvector

Description

Store images in buffers.

#include <graph.h>

void _far _getimage(short *x1*, **short** *y1*, **short** *x2*, **short** *y2*, **char _huge** **image* **);**

void _far _getimage_w(double *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2*,
char _huge **image* **);**

void _far _getimage_wxy(struct_wxycoord _far **pwxy1*,
struct_wxycoord _far **pwxy2*, **char _huge** **image* **);**

x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
image	Storage buffer for screen image

Remarks

The **_getimage** functions store the screen image defined by a specified bounding rectangle into the buffer pointed to by *image*.

The **_getimage** function defines the bounding rectangle with the view coordinates (*x1*, *y1*) and (*x2*, *y2*).

The **_getimage_w** function defines the bounding rectangle with the window coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*).

The **_getimage_wxy** function defines the bounding rectangle with the window-coordinate pairs *pwxy1* and *pwxy2*.

The buffer must be large enough to hold the image. You can determine the size by calling the appropriate **_imagesize** function at run time, or by using the formula described on the **_imagesize** reference page.

Return Value

None. Use **_grstatus** to check success.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also _grstatus, _imagesize functions, _putimage functions

Example

```
/* GIMAGE.C: This example illustrates animation routines including:
 *          _imagesize     _getimage     _putimage
 */

#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <malloc.h>
#include <graph.h>

short action[5]  = { _GPSET,   _GPRESET, _GXOR,    _GOR,     _GAND     };
char *descrip[5] = {  "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   " };

void exitfree( char _huge *buffer );

void main()
{
    char  _huge *buffer;   /* Far pointer (with _fmalloc) could be used. */
    long  imsize;
    short i, x, y = 30;

    if( !_setvideomode( _MAXRESMODE ) )
        exit( 1 );

    /* Measure the image to be drawn and allocate memory for it. */
    imsize = (size_t)_imagesize( -16, -16, +16, +16 );
    buffer = halloc( imsize, sizeof( char ) );
    if ( buffer == (char _far *)NULL )
        exit( 1 );

    _setcolor( 3 );
    for ( i = 0; i < 5; i++ )
    {
        /* Draw ellipse at new position and get a copy of it. */
        x = 50; y += 40;
        _ellipse( _GFILLINTERIOR, x - 15, y - 15, x + 15, y + 15 );
        _getimage( x - 16, y - 16, x + 16, y + 16, buffer );
        if( _grstatus() )
            exitfree( buffer );        /* Quit on error                  */
```

```
        /* Display action type and copy a row of ellipses with that type. */
        _settextposition( 1, 1 );
        _outtext( descrip[i] );
        while( x < 260 )
        {
            x += 5;
            _putimage( x - 16, y - 16, buffer, action[i] );
            if( _grstatus() < 0 )      /* Ignore warnings, quit on errors. */
                exitfree( buffer );
        }
        getch();
    }
    exitfree( buffer );
}

void exitfree( char _huge *buffer )
{
    hfree( buffer );
    exit( !_setvideomode( _DEFAULTMODE ) );
}
```

Description Gets the current line style.

#include <graph.h>

unsigned short _far _getlinestyle(void);

Remarks Some graphics routines (**_lineto**, **_polygon**, 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. The mask is a 16-bit array in which 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.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_lineto** functions, **_polygon** functions, **_rectangle** functions, **_setlinestyle**, **_setwritemode**

Example

```
/* GLINESTY.C: This program illustrates _setlinestyle and _getlinestyle. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void zigzag( short x1, short y1, short size );

void main()
{

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );

   /* Set line style and draw rectangle. */
   _setlinestyle( 0x4d );
   _rectangle( _GBORDER, 10, 10, 60, 60 );
   getch();
```

```
   /* Draw figure with function that changes and restores line style. */
   zigzag( 100, 100, 90 );
   getch();

   /* Original style reused. */
   _rectangle( _GBORDER, 190, 190, 130, 130 );
   getch();

   _setvideomode( _DEFAULTMODE );
}

/* Draw box with changing line styles. Restore original style. */
void zigzag( short x1, short y1, short size )
{
   short x, y, oldcolor;
   unsigned short oldstyle;
   unsigned short style[16] = { 0x0001, 0x0003, 0x0007, 0x000f,
                                0x001f, 0x003f, 0x007f, 0x00ff,
                                0x01ff, 0x03ff, 0x07ff, 0x0fff,
                                0x1fff, 0x3fff, 0x7fff, 0xffff };

   oldcolor = _getcolor();
   oldstyle = _getlinestyle();            /* Save old line style.        */
   for( x = 3, y = 3; x < size; x += 3, y += 3 )
   {
      _setcolor( x % 16 );
      _setlinestyle( style[x % 16] );     /* Set and use new line styles */
      _rectangle( _GBORDER, x1 - x, y1 - y, x1 + x, y1 + y );
   }
   _setlinestyle( oldstyle );             /* Restore old line style.     */
   _setcolor( oldcolor );
}
```

Description

Gets physical coordinates.

#include <graph.h>

struct xycoord _far _getphyscoord(short *x*, short *y*);

x, *y*	View coordinates to translate

Remarks

The **_getphyscoord** function translates the view coordinates (*x*, *y*) to physical coordinates and returns them in an **xycoord** structure, defined in GRAPH.H.

The **xycoord** structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

Return Value

None.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getviewcoord functions, **_grstatus**, **_setvieworg**, **_setviewport**

Example

See the example for **_setwindow**.

Description Gets the process identification.

#include <process.h> Required only for function declarations

int getpid(void);

Remarks The **getpid** function returns the process ID, an integer that uniquely identifies the calling process.

Return Value The **getpid** function returns the process ID. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **mktemp**

Example

```
/* GETPID.C: This program uses getpid to obtain the process ID and
 * then prints the ID.
 */

#include <stdio.h>
#include <process.h>

void main( )
{
   /* If run from DOS, shows different ID for DOS than for DOS shell.
    * If execed or spawned, shows ID of parent.
    */
   printf( "\nProcess id of parent: %d\n", getpid() );
}
```

Output

```
Process id of parent: 828
```

Description Get pixel values.

#include <graph.h>

short _far _getpixel(short *x*, **short** *y* **);**

short _far _getpixel_w(double *wx*, **double** *wy* **);**

x, *y*	Pixel position
wx, *wy*	Pixel position

Remarks The functions in the **_getpixel** family return the pixel value (a color index) at a specified location. The **_getpixel** function uses the view coordinate (*x*, *y*). The **_getpixel_w** function uses the window coordinate (*wx*, *wy*). The range of possible pixel values is determined by the current video mode. The color translation of pixel values is determined by the current palette.

Return Value If successful, the function returns the color index. If the function fails (for example, the point lies outside the clipping region, or the program is in a text mode), it returns –1.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getvideoconfig**, **_grstatus**, **_remapallpalette**, **_remappalette**, **_selectpalette**, **_setpixel** functions, **_setvideomode**

Example

```
/* GPIXEL.C: This program assigns different colors to randomly
 * selected pixels.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   short xvar, yvar;
   struct videoconfig vc;
```

```
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );

   /* Draw filled ellipse to turn on certain pixels. */
   _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                             vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );

   /* Draw random pixels in random colors... */
   while( !kbhit() )
   {
      /* ...but only if they are already on (inside the ellipse). */
      xvar = rand() % vc.numxpixels;
      yvar = rand() % vc.numypixels;
      if( _getpixel( xvar, yvar ) != 0 )
      {
         _setcolor( rand() % 16 );
         _setpixel( xvar, yvar );
      }
   }

   getch();          /* Throw away the keystroke. */
   _setvideomode( _DEFAULTMODE );
}
```

Description Gets a line from the **stdin** stream.

#include <stdio.h>

char *gets(char ***_buffer_ **);

buffer	Storage location for input string

Remarks 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 newline character (**\n**). The **gets** function then replaces the newline character with a null character (**'\0'**) before returning the line. In contrast, the **fgets** function retains the newline 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fgets, fputs, puts**

Example

```
/* GETS.C */

#include <stdio.h>

void main()
{
   char line[81];

   printf( "Input a string: " );
   gets( line );
   printf( "The line entered was: %s\n", line );
}
```

Output

```
Input a string: This is a string
The line entered was: This is a string
```

Description

Gets the current text color.

#include <graph.h>

short _far _gettextcolor(void);

Remarks

The **_gettextcolor** function returns the color index of the current text color. The text color is set by the **_settextcolor** function and affects text output with the **_outtext** and **_outmem** functions only. The **_setcolor** function sets the color for font text output using the **_outgtext** function.

The default is 7 in test modes; it is the highest legal color index of the current palette in graphics modes.

Return Value

The **_gettextcolor** function returns the color index of the current text color.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_getvideoconfig, _remappalette, _selectpalette, _setcolor, _settextcolor

Example

See the example for **_gettextposition.**

Description Gets the current cursor attribute.

#include <graph.h>

short _far _gettextcursor(void);

Remarks The **_gettextcursor** function returns the current cursor attribute (i.e., the shape). This function works only in text video modes.

Return Value The function returns the current cursor attribute, or –1 if an error occurs (such as a call to the function in a graphics mode).

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_displaycursor, _grstatus, _settextcursor**

Example See the example for **_settextcursor**.

Description

Gets the current text position.

#include <graph.h>

struct rccoord _far _gettextposition(void);

Remarks

The **_gettextposition** function returns the current text position as an **rccoord** structure, defined in GRAPH.H.

The **rccoord** structure contains the following elements:

Element	Description
short row	Row coordinate
short col	Column coordinate

Remarks

The text position given by the coordinates (1,1) is defined as the upper-left corner of the text window.

Text output from the **_outtext** and **_outmem** functions begins at the current text position. Font text is not affected by the current text position. Font text output begins at the current graphics output position, which is a separate position.

Return Value

None.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_getcurrentposition functions, **_moveto** functions, **_outmem**, **_outtext**, **_settextposition**, **_settextwindow**, **_wrapon**

Example

```
/* OUTTXT.C: This example illustrates text output functions:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition
 */

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

```
      char buffer [80];

void main()
{

   /* Save original foreground, background, and text position. */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct rccoord oldpos;

   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );

   /* First time no blink, second time blinking. */
   for( blink = 0; blink <= 16; blink += 16 )
   {
      /* Loop through 8 background colors. */
      for( bgd = 0; bgd < 8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf(buffer, "Back: %d Fore:", bgd );
         _outtext( buffer );

         /* Loop through 16 foreground colors. */
         for( fgd = 0; fgd < 16; fgd++ )
         {
            _settextcolor( fgd + blink );
            sprintf( buffer, " %2d ", fgd + blink );
            _outtext( buffer );
         }
      }
   }
   getch();

   /* Restore original foreground, background, and text position. */
   _settextcolor( oldfgd );
   _setbkcolor( oldbgd );
   _clearscreen( _GCLEARSCREEN );
   _settextposition( oldpos.row, oldpos.col );
}
```

Description Gets the boundaries of the current text window.

#include <graph.h>

void _far _gettextwindow(short _far ****r1*, short _far ****c1***, short _far ****r2***,**
short _far ****c2*);**

r1	Top row of current text window
c1	Leftmost column of current text window
r2	Bottom row of current text window
c2	Rightmost column of current text window

Remarks The **_gettextwindow** function finds the boundaries of the current text window. The text window is the region of the screen to which output from the **_outtext** and **_outmem** functions is limited. By default, this is the entire screen, unless it has been redefined by the **_settextwindow** function.

The window defined by **_settextwindow** has no effect on output from the **_outgtext** function. Text displayed with **_outgtext** is limited to the current viewport.

Return Value None.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_gettextposition, _outmem, _outtext, _scrolltextwindow, _settextposition, _settextwindow, _wrapon**

Example See the example for **_scrolltextwindow**.

Description Gets graphics video configuration information.

#include <graph.h>

struct videoconfig _far * _far _getvideoconfig(struct videoconfig _far ***config*** **);**

config	Configuration information

Remarks The **_getvideoconfig** function returns the current graphics environment configuration in a **videoconfig** structure, defined in GRAPH.H.

The values returned reflect the currently active video adapter and monitor, as well as the current video mode.

The **videoconfig** structure contains the following members, each of which is of type **short**:

Member	Contents
adapter	Active display adapter
bitsperpixel	Number of bits per pixel
memory	Adapter video memory in kilobytes
mode	Current video mode
monitor	Active display monitor
numcolors	Number of color indices
numtextcols	Number of text columns available
numtextrows	Number of text rows available
numvideopages	Number of available video pages
numxpixels	Number of pixels on the *x* axis
numypixels	Number of pixels on the *y* axis

The values for the **adapter** member of the **videoconfig** structure are given by the manifest constants shown in the list below. For any applicable adapter (_CGA, _EGA, or _VGA), the corresponding Olivetti® adapter (_OCGA, _OEGA, or _OVGA) represents a superset of graphics capabilities.

Adapter Constant	Meaning
_CGA	Color Graphics Adapter
_EGA	Enhanced Graphics Adapter
_HGC	Hercules® Graphics Card
_MCGA	Multicolor Graphics Array
_MDPA	Monochrome Display Printer Adapter
_OCGA	Olivetti (AT&T®) Color Graphics Adapter
_OEGA	Olivetti (AT&T) Enhanced Graphics Adapter
_OVGA	Olivetti (AT&T) Video Graphics Array
_VGA	Video Graphics Array

The values for the **monitor** member of the **videoconfig** structure are given by the manifest constants listed below:

Monitor Constant	Meaning
_ANALOG	Analog monochrome and color
_ANALOGCOLOR	Analog color only
_ANALOGMONO	Analog monochrome only
_COLOR	Color (or enhanced monitor emulating a color monitor)
_ENHCOLOR	Enhanced color
_MONO	Monochrome monitor

In every text mode, including monochrome, the **_getvideoconfig** function returns the value 32 for the number of available colors. The value 32 indicates the range of values (0–31) accepted by the **_settextcolor** function. This includes 16 normal colors (0–15) and 16 blinking colors (16–31). Blinking is selected by adding 16 to the normal color index. Because monochrome text mode has fewer unique display attributes, some color indices are redundant. However, because blinking is selected in the same manner, monochrome text mode has the same range (0–31) as other text modes.

Return Value The **_getvideoconfig** function returns the video configuration information in a structure, as noted above. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_setvideomode, _setvideomoderows**

Example

```
/* GVIDCFG.C: This program displays information about the current
 * video configuration.
 */

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

void main()
{
   struct videoconfig vc;
   short  c;
   char   b[500];                         /* Buffer for string */

   _getvideoconfig( &vc );

   /* Write all information to a string, then output string. */
   c  = sprintf( b,     "X pixels:     %d\n", vc.numxpixels );
   c += sprintf( b + c, "Y pixels:     %d\n", vc.numypixels );
   c += sprintf( b + c, "Text columns: %d\n", vc.numtextcols );
   c += sprintf( b + c, "Text rows:    %d\n", vc.numtextrows );
   c += sprintf( b + c, "Colors:       %d\n", vc.numcolors );
   c += sprintf( b + c, "Bits/pixel:   %d\n", vc.bitsperpixel );
   c += sprintf( b + c, "Video pages:  %d\n", vc.numvideopages );
   c += sprintf( b + c, "Mode:         %d\n", vc.mode );
   c += sprintf( b + c, "Adapter:      %d\n", vc.adapter );
   c += sprintf( b + c, "Monitor:      %d\n", vc.monitor );
   c += sprintf( b + c, "Memory:       %d\n", vc.memory );
   _outtext( b );
}
```

Output

```
X pixels:     0
Y pixels:     0
Text columns: 80
Text rows:    25
Colors:       32
Bits/pixel:   0
Video pages:  1
Mode:         3
Adapter:      8
Monitor:      24
Memory:       256
```

Description

Translate coordinates to view coordinates.

#include <graph.h>

struct xycoord _far _getviewcoord(short *x*, **short** *y* **);**

struct xycoord _far _getviewcoord_w(double *wx*, **double** *wy* **);**

struct xycoord _far _getviewcoord_wxy(struct _wxycoord _far **pwxy1* **);**

x, *y*	Physical point to translate
wx, *wy*	Window point to translate
pwxy1	Window point to translate

Remarks

The **_getviewcoord** routines translate the specified coordinates (*x*, *y*) from one coordinate system to view coordinates and then return them in an **xycoord** structure, defined in GRAPH.H. The **xycoord** structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The various **_getviewcoord** routines translate in the following manner:

Routine	Translation
_getviewcoord	Physical coordinates (*x*, *y*) to view coordinates
_getviewcoord_w	Window coordinates (*wx*, *wy*) to view coordinates
_getviewcoord_wxy	Window coordinates structure (*pwxy1*) to view coordinates

C 5.1 Version Difference *In Microsoft C version 5.1, the function* ***_getviewcoord*** *was called* ***_getlogcoord.***

Return Value

The **_getviewcoord** function returns the coordinates as noted above. There is no error return.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getphyscoord, _getwindowcoord, _grstatus**

Example See the example for **_setwindow**.

Description Gets the current visual page number.

#include <graph.h>

short _far _getvisualpage(void);

Remarks The **_getvisualpage** function returns the current visual page number.

Return Value The function returns the number of the current visual page. All hardware combinations support at least one page (page number 0). In OS/2, only page 0 is available.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_getactivepage, _gettextcolor, _gettextposition, _outtext, _setactivepage, _settextcolor, _settextposition, _settextwindow, _setvideomode, _setvisualpage, _wrapon**

Example See the example for **_getactivepage**.

Description

Gets an integer from a stream.

#include <stdio.h>

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

stream	Pointer to **FILE** structure

Remarks

The **getw** function reads the next binary value of type **int** from the file associated with *stream* and increments 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, since the **EOF** value is also a legitimate integer value, **feof** or **ferror** should be used to verify an end-of-file or error condition.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

The **getw** function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with **getw**, since the size of the **int** type and the ordering of bytes within the **int** type differ across systems.

See Also

putw

Example

```
/* GETW.C: This program uses getw to read a word from a stream,
 * then performs an error check.
 */

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

void main()
{
   FILE *stream;
   int i;
```

```
   if( (stream = fopen( "getw.c", "rb" )) == NULL )
      printf( "Couldn't open file\n" );
   else
   {
      /* 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( "First data word in file: 0x%.4x\n", i );
      fclose( stream );
   }
}
```

Output

```
First data word in file: 0x2a2f
```

Description Translates view coordinates to window coordinates.

#include <graph.h>

struct _wxycoord _far _getwindowcoord(short *x*, **short** *y* **);**

x, *y*	Physical point to translate

Remarks The **_getwindowcoord** function translates the view coordinates (*x*, *y*) to window coordinates and returns them in the **_wxycoord** structure, defined in GRAPH.H.

The **_wxycoord** structure contains the following elements:

Element	Description
double wx	*x* coordinate
double wy	*y* coordinate

Return Value The function returns the coordinates in the **_wxycoord** structure. There is no error return.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getphyscoord**, **_getviewcoord** functions, **_moveto** functions, **_setwindow**

Example See the example for **_setwindow**.

Description

Gets the current logical mode for line drawing.

#include <graph.h>

short _far _getwritemode(void);

Remarks

The **_getwritemode** function returns the current logical write mode, which is used when drawing lines with the **_lineto**, **_polygon**, and **_rectangle** functions.

The default value is **_GPSET**, which causes lines to be drawn in the current graphics color. The other possible return values are **_GXOR**, **_GAND**, **_GOR**, and **_GPRESET**. See **_putimage** for more details on these manifest constants.

Return Value

The **_getwritemode** function returns the current logical write mode, or –1 if not in graphics mode.

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

_grstatus, **_lineto** functions, **_putimage** functions, **_rectangle** functions, **_setcolor**, **_setlinestyle**, **_setwritemode**

Example

```
/* GWRMODE.C: This program illustrates _getwritemode and _setwritemode. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

short wmodes[5]  = { _GPSET,   _GPRESET, _GXOR,    _GOR,     _GAND    };
char *wmstr[5]   = { "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   " };

void box( short x, short y, short size, short writemode, short fillmode );

void main()
{
   short i, x, y;
```

```
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );

   x = y = 70;
   box( x, y, 50, _GPSET, _GFILLINTERIOR );
   _setcolor( 2 );
   box( x, y, 40, _GPSET, _GFILLINTERIOR );
   for( i = 0; i < 5; i++ )
   {
      _settextposition( 1, 1 );
      _outtext( wmstr[i] );
      box( x += 12, y += 12, 50, wmodes[i], _GBORDER );
      getch();
   }
   _setvideomode( _DEFAULTMODE );
}

void box( short x, short y, short size, short writemode, short fillmode )
{
    short wm, side;

    wm = _getwritemode();            /* Save write mode and set new. */
    _setwritemode( writemode );
    _rectangle( fillmode, x - size, y - size, x + size, y + size );
    _setwritemode( wm );             /* Restore original write mode. */
}
```

Description

Converts a time value to a structure.

#include <time.h>

struct tm *gmtime(const time_t ***timer* **);

timer	Pointer to stored time

Remarks

The **gmtime** function converts the *timer* value to a structure. The *timer* argument represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time. This value is usually obtained from a call to the **timer** function.

The **gmtime** function breaks down the *timer* value and stores it in a structure of type **tm**, defined in TIME.H. (See **asctime** for a description of the structure members.) The structure result reflects Greenwich mean time, not local time.

The fields of the structure type **tm** store the following values, each of which is an **int**:

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**

The **gmtime, mktime,** and **localtime** functions use a single statically allocated structure to hold the result. Each call to one of these routines destroys the result of any previous call.

DOS and OS/2 do not accommodate dates prior to 1980. If *timer* represents a date prior to 1980, **gmtime** returns **NULL.**

Return Value

The **gmtime** function returns a pointer to the structure result. There is no error return.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also asctime, ctime, ftime, localtime, time

Example

```
/* GMTIME.C: 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.
 */

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

void main()
{
   struct tm *newtime;
   long ltime;

   time( &ltime );

   /* Obtain Greenwich mean time: */
   newtime = gmtime( &ltime );
   printf( "Greenwich mean time is %s\n", asctime( newtime ) );
}
```

Output

```
Greenwich mean time is Fri Jun 16 16:37:53 1989
```

Description

Returns the status of the most recent graphics function call.

#include <graph.h>

short _far _grstatus(void);

Remarks

The **_grstatus** function returns the status of the most recently used graphics function. The **_grstatus** function can be used immediately following a call to a graphics routine to determine if errors or warnings were generated. Return values less than 0 are errors, and values greater than 0 are warnings.

The following manifest constants are defined in the GRAPH.H header file for use with the **_grstatus** function:

Value	Constant	Meaning
0	**_GROK**	Success
–1	**_GRERROR**	Graphics error
–2	**_GRMODENOTSUPPORTED**	Requested video mode not supported
–3	**_GRNOTINPROPERMODE**	Requested routine only works in certain video modes
–4	**_GRINVALIDPARAMETER**	One or more parameters invalid
–5	**_GRFONTFILENOTFOUND**	No matching font file found
–6	**_GRINVALIDFONTFILE**	One or more font files invalid
–7	**_GRCORRUPTEDFONTFILE**	One or more font files inconsistent
–8	**_GRINSUFFICIENTMEMORY**	Not enough memory to allocate buffer or to complete a **_floodfill** operation
–9	**_GRINVALIDIMAGEBUFFER**	Image buffer data inconsistent
1	**_GRMOOUTPUT**	No action taken
2	**_GRCLIPPED**	Output was clipped to viewport
3	**_GRPARAMETERALTERED**	One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order

After a graphics call, use an **if** statement to compare the return value of **_grstatus** to _GROK. For example:

```
if( _grstatus < _GROK )
    /*handle graphics error*/ ;
```

The functions listed below cannot cause errors, and they all set **_grstatus** to **GROK:**

_displaycursor	**_gettextposition**	**_outmem**
_getactivepage	**_gettextwindow**	**_outtext**
_getgtextvector	**_getvideoconfig**	**_unregisterfonts**
_gettextcolor	**_getvisualpage**	**_wrapon**

See the list below for the graphics functions that affect **_grstatus.** The list shows error or warning messages that can be set by the graphics function. In addition to the error codes listed, all of these functions can produce the **_GRERROR** error code.

Function	Possible _grstatus Error Codes	Possible _grstatus Warning Codes
_arc functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	_GRNOOUTPUT, _GRCLIPPED
_clearscreen	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	
_ellipse functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_getarcinfo	_GRNOTINPROPERMODE	
_getcurrentposition functions	_GRNOTINPROPERMODE	
_getfontinfo	(_GRERROR only)	
_getgtextextent	(_GRERROR only)	
_getgtextvector	_GRPARAMETERALTERED	
_getimage	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_getphyscoord	_GRNOTINPROPERMODE	
_getpixel	_GRNOTINPROPERMODE	
_gettextcursor	_GRNOTINPROPERMODE	
_getviewcoord functions	_GRNOTINPROPERMODE	

Continued on next page

Function	Possible _grstatus Error Codes	Possible _grstatus Warning Codes
_getwindowcoord	_GRNOTINPROPERMODE	
_getwritemode	_GRNOTINPROPERMODE	
_imagesize functions	_GRNOTINPROPERMODE	
_lineto functions	_GRNOTINPROPERMODE	_GRNOOUTPUT, _GRCLIPPED
_moveto functions	_GRNOTINPROPERMODE	
_outgtext	_GRNOTINPROPERMODE	_GRCLIPPED, _GRNOOUTPUT
_pie functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_polygon functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_putimage functions	_GRERROR, _GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINVALIDIMAGEBUFFER	_GRPARAMETERALTERED, _GRNOOUTPUT
_rectangle functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_registerfonts	_GRCORRUPTEDFONTFILE, _GRFONTFILENOTFOUND, _GRINSUFFICIENTMEMORY, _GRINVALIDFONTFILE	
_scrolltextwindow		_GRNOOUTPUT
_selectpalette	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	
_setactivepage	_GRINVALIDPARAMETER	
_setbkcolor	_GRINVALIDPARAMETER	_GRPARAMETERALTERED
_setcliprgn	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setcolor	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setfont	_GRERROR, _GRFONTFILENOTFOUND, _GRINSUFFICIENTMEMORY, _GRPARAMETERALTERED	

Continued on next page

Function	Possible _grstatus Error Codes	Possible _grstatus Warning Codes
_setgtextvector	_GRPARAMETERALTERED	
_settextcolor		_GRPARAMETERALTERED
_settextcursor	_GRNOTINPROPERMODE	
_settextposition		_GRPARAMETERALTERED
_settextrows	_GRINVALIDPARAMETER	_GRPARAMETERALTERED
_settextwindow		_GRPARAMETERALTERED
_setvideomode	_GRERROR, _GRMODENOTSUPPORTED, _GRINVALIDPARAMETER	
_setvideomoderows	_GRERROR, _GRMODENOTSUPPORTED, _GRINVALIDPARAMETER	
_setvieworg	_GRNOTINPROPERMODE	
_setviewport	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setvisualpage	_GRINVALIDPARAMETER	
_setwindow	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	_GRPARAMETERALTERED
_setwritemode	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	

Return Value The **_grstatus** function returns the status of the most recently used graphics function.

See Also **_arc** functions, **_ellipse** functions, **_floodfill**, **_lineto** functions, **_pie** functions, **_remapallpalette**, **_setactivepage**, **_setbkcolor**, **_setcolor**, **_setpixel** functions, **_settextcolor**, **_settextcursor**, **_setvisualpage**, **_setwindow**, **_setwritemode**

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

halloc

Description Allocates a huge memory block.

#include <malloc.h> Required only for function declarations

void _huge *halloc(long *num*, **size_t** *size* **);**

num	Number of elements
size	Length in bytes of each element

Remarks The **halloc** function allocates a huge array from the operating system consisting of *num* 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), the size of an array element must then be a power of 2.

Return Value The **halloc** function returns a **void huge** 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 **void huge**, use a type cast on the return value. If the request cannot be satisfied, the return value is **NULL.**

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **calloc** functions, **free** functions, **hfree**, **malloc** functions

Example

```
/* HALLOC.C: This program uses halloc to allocate space for 30,000 long
 * integers, then uses hfree to deallocate the memory.
 */

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

void main()
{
   long _huge *hbuf;
```

```
   /* Allocate huge buffer */
   hbuf = (long _huge *)halloc( 30000L, sizeof( long ) );
   if ( hbuf == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory successfully allocated\n" );

   /* Free huge buffer */
   hfree( hbuf );
}
```

Output

```
Memory successfully allocated
```

Description

Handle critical error conditions.

#include <dos.h>

void _harderr(void(_far ***handler*** **)( ));**

void _hardresume(int ***result*** **);**

void _hardretn(int ***error*** **);**

handler ( )	New INT 0x24 handler
result	Handler return parameter
error	Error to return from

Remarks

These three functions are used to handle critical error conditions that use DOS interrupt 0x24. The **_harderr** function installs a new critical-error handler for interrupt 0x24.

The **_hardresume** and **_hardreturn** functions control how the program will return from the new critical-error handler installed by **_harderr**. The **_hardresume** function returns to DOS from a user-installed critical-error handler, and the **_hardreturn** function returns directly to the application program from a user-installed critical-error handler.

The **_harderr** function does not directly install the handler pointed to by *handler*; instead, **_harderr** installs a handler that calls the function referenced by *handler*. 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. It contains the AX register value passed by DOS to the INT 0x24 handler. The *errcode* argument is the DI register value that 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	Unknown unit
2	Drive not ready
3	Unknown command
4	Cyclic-redundancy-check 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
10	Write fault
11	Read fault
12	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.

Errors on Disk Devices

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:

Bit	Meaning
15	Disk error if false (0).
14	Not used.
13	"Ignore" response not allowed if false (0).
12	"Retry" response not allowed if false (0).
11	"Fail" response not allowed if false (0). Note that DOS changes "fail" to "abort".
10, 9	**Code** / **Location** **00** DOS **01** File allocation table **10** Directory **11** Data area
8	Read error if false; write error if true.

The low-order byte of *deverror* indicates the drive in which the error occurred (0 = drive A, 1 = drive B, etc.).

Errors on Other Devices

If the error occurs on a device other than a disk drive, the high-order bit (bit 15) of the *deverror* argument is 1. The attribute word located at offset 4 in the device-header block indicates the type of device that 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 1, the error occurred on a character device and bits 0–3 of the attribute word indicate the type of device, as shown in the following list:

Bit	Meaning
0	Current standard input
1	Current standard output
2	Current null device
3	Current clock device

Restrictions on Handler Functions

The user-defined handler function can issue only system calls 0x01 through 0x0C, 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 can be used to obtain further information about the error that occurred.

Using _hardresume and _harderr

If the handler returns, it can do so

1. From the **return** statement
2. From the **_hardresume** function
3. From the **_hardretn** function

If the handler returns from **_hardresume** or from a **return** statement, the handler returns to DOS.

The **_hardresume** function should be called only from within the user-defined hardware error-handler function. The result supplied to **_hardresume** must be one of the following constants:

Constant	Action
_HARDERR_ABORT	Abort the program by issuing INT 0x23
_HARDERR_FAIL	Fail the system call that is in progress (this is not supported on DOS 2.x)
_HARDERR_IGNORE	Ignore the error
_HARDERR_RETRY	Retry the operation

The **_hardretn** function allows the user-defined hardware error handler to return directly to the application program rather than returning to DOS. The application resumes at the point just after the failing I/O function request. The **_hardretn** function should be called only from within a user-defined hardware error-handler function.

The error parameter of **_hardretn** should be a DOS error code, as opposed to the XENIX-style error code that is available in **errno**. Refer to *MS-DOS Encyclopedia* (Duncan, ed.; Redmond, Wa.: Microsoft Press, 1988) or *Programmer's PC Sourcebook* (Hogan; Redmond, Wa.: Microsoft Press, 1988) for information about the DOS error codes that may be returned by a given DOS function call.

If the failing I/O function request is an INT 0x21 function greater than or equal to function 0x38, **_hardretn** will then 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 (which 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.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_chain_intr, _dos_getvect, _dos_setvect**

Description Add memory to the heap (**_heapadd**) or to the based heap (**_bheapadd**).

#include <malloc.h> Required only for function declarations

int _heapadd(void _far ****memblock*, **size_t** *size* **);**

int _bheapadd(_segment *seg*, **void _based (void)** ****memblock*, **size_t** *size* **);**

seg	Based-heap segment selector
buffer	Pointer to heap memory
size	Size in bytes of memory to add

Remarks The **_heapadd** and **_bheapadd** functions add an unused piece of memory to the heap. The **_bheapadd** function adds the memory to the based heap specified by *seg*. The **_heapadd** function looks at the segment value and, if it is DGROUP, adds the memory to the near heap. Otherwise, **_heapadd** adds the memory to the far heap.

Return Value These functions return 0 if successful, or –1 if an error occurred.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **free** functions, **halloc**, **hfree**, **malloc** functions, **realloc** functions

Example

```
/* HEAPMIN.C: This program illustrates heap management using
 * _heapadd and _heapmin.
 */

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

```
void heapdump( char *msg );      /* Prototype */

char s1[] = { "Here are some strings that we use at first, then don't\n" };
char s2[] = { "need any more. We'll give their space to the heap.\n" };

void main()
{
    int *p[3], i;

    printf( "%s%s", s1, s2 );
    heapdump( "Initial heap" );

    /* Give space of used strings to heap. */
    _heapadd( s1, sizeof( s1 ) );
    _heapadd( s2, sizeof( s2 ) );
    heapdump( "After adding used strings" );

    /* Allocate some blocks. Some may use string blocks from _heapadd. */
    for( i = 0; i < 2; i++ )
        if( (p[i] = (int *)calloc( 10 * (i + 1), sizeof( int ) )) == NULL )
        {
            --i;
            break;
        }
    heapdump( "After allocating memory" );

    /* Free some of the blocks. */
    free( p[1] );
    free( p[2] );
    heapdump( "After freeing memory" );

    /* Minimize heap. */
    _heapmin();
    heapdump( "After compacting heap" );
}

/* Walk through heap entries, displaying information about each block. */
void heapdump( char *msg )
{
    struct _heapinfo hi;

    printf( "%s\n", msg );
    hi._pentry = NULL;
    while( _heapwalk( &hi ) == _HEAPOK )
        printf( "\t%s block at %Fp of size %u\t\n",
                hi._useflag == _USEDENTRY ? "USED" : "FREE",
                hi._pentry,
                hi._size );
    getch();
}
```

Output

```
Here are some strings that we use at first, then don't
need any more. We'll give their space to the heap.
Initial heap
   USED block at 2D39:0E9C of size 364
   USED block at 2D39:100A of size 36
   USED block at 2D39:1030 of size 512
   FREE block at 2D39:1232 of size 460
After adding used strings
   FREE block at 2D39:0044 of size 52
   FREE block at 2D39:007A of size 50
   USED block at 2D39:00AE of size 3564
   USED block at 2D39:0E9C of size 364
   USED block at 2D39:100A of size 36
   USED block at 2D39:1030 of size 512
   FREE block at 2D39:1232 of size 460
After allocating memory
   USED block at 2D39:0044 of size 20
   USED block at 2D39:005A of size 40
   FREE block at 2D39:0084 of size 40
   USED block at 2D39:00AE of size 3564
   USED block at 2D39:0E9C of size 364
   USED block at 2D39:100A of size 36
   USED block at 2D39:1030 of size 512
   FREE block at 2D39:1232 of size 460
After freeing memory
   USED block at 2D39:0044 of size 20
   FREE block at 2D39:005A of size 40
   FREE block at 2D39:0084 of size 40
   USED block at 2D39:00AE of size 3564
   USED block at 2D39:0E9C of size 364
   USED block at 2D39:100A of size 36
   USED block at 2D39:1030 of size 512
   FREE block at 2D39:1232 of size 460
After compacting heap
   USED block at 2D39:0044 of size 20
   FREE block at 2D39:005A of size 82
   USED block at 2D39:00AE of size 3564
   USED block at 2D39:0E9C of size 364
   USED block at 2D39:100A of size 36
   USED block at 2D39:1030 of size 512
   FREE block at 2D39:1232 of size 12
```

Description

Run consistency checks on the heap.

#include <malloc.h>

int _heapchk(void);

int _bheapchk(_segment *seg* **);**

int _fheapchk(void);

int _nheapchk(void);

seg	Specified base heap

Remarks

The **_heapchk** routines help to debug heap-related problems by checking for minimal consistency of the heap.

Each function checks a particular heap, as listed below:

Function	Heap Checked
_heapchk	Depends on data model of program
_bheapchk	Based heap specified by *seg* value
_fheapchk	Far heap (outside the default data segment)
_nheapchk	Near heap (inside the default data segment)

In large data models (that is, compact-, large-, and huge-model programs), **_heapchk** maps to **_fheapchk**. In small data models (tiny-, small-, and medium-model programs), **_heapchk** maps to **_nheapchk**.

Return Value

All four routines return an integer value that is one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	Initial header information cannot be found, or it is bad
_HEAPBADNODE	Bad node has been found, or the heap is damaged
_HEAPEMPTY	Heap has not been initialized
_HEAPOK	Heap appears to be consistent

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_heapset** functions, **_heapwalk** functions

Example

```
/* HEAPCHK.C: This program checks the heap for consistency
 * and prints an appropriate message.
 */

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

void main()
{
   int  heapstatus;
   char *buffer;

   /* Allocate and deallocate some memory */
   if( (buffer = (char *)malloc( 100 )) != NULL )
      free( buffer );

   /* Check heap status */
   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" );
         break;
      case _HEAPBADNODE:
         printf( "ERROR - bad node in heap\n" );
         break;
   }
}
```

Output

```
 OK - heap is fine
```

Description

Release unused heap memory to the operating system.

#include <malloc.h>

int _heapmin(void);

int _bheapmin(_segment *seg* **)**

int _fheapmin(void);

int _nheapmin(void);

seg	Specified based-heap selector

Remarks

The **_heapmin** functions minimize the heap by releasing unused heap memory to the operating system.

The various **_heapmin** functions release unused memory in these heaps:

Function	Heap Minimized
_heapmin	Depends on data model of program
_bheapmin	Based heap specified by *seg* value; **_NULLSEG** specifies all based heaps
_fheapmin	Far heap (outside default data segment)
_nheapmin	Near heap (inside default data segment)

In large data models (that is, compact-, large-, and huge-model programs), **_heapmin** maps to **_fheapmin.** In small data models (tiny-, small-, and medium-model programs), **_heapmin** maps to **_nheapmin.**

Based-heap segments are never freed (i.e., unlinked from the based heap list and released back to the operating system) by the **_bheapmin** function. The **_bfreeseg** function is used for that purpose.

Return Value

The **_heapmin** functions return 0 if the function completed successfully, or –1 in the case of an error.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_bfreeseg, free functions, **malloc** functions

Description Check heaps for minimal consistency and set the free entries to a specified value.

#include <malloc.h>

int _heapset(unsigned int *fill* **);**

int _bheapset(_segment *seg*, **unsigned int** *fill* **);**

int _fheapset(unsigned int *fill* **);**

int _nheapset(unsigned int *fill* **);**

fill	Fill character
seg	Specified based-heap segment selector

Remarks The **_heapset** family of routines helps debug heap-related problems in programs by showing free memory locations or nodes unintentionally overwritten.

The **_heapset** routines first check for minimal consistency on the heap in a manner identical to that of the **_heapchk** functions. After the consistency check, the **_heapset** functions set each byte of the heap's free entries to the *fill* value. This known value shows which memory locations of the heap contain free nodes and which locations contain data that were unintentionally written to freed memory.

The various **_heapset** functions check and fill these heaps:

Function	Heap Filled
_heapset	Depends on data model of program
_bheapset	Based heap specified by *seg* value; **_NULLSEG** specifies all based heaps
_fheapset	Far heap (outside default data segment)
_nheapset	Near heap (inside default data segment)

In large data models (that is, compact-, large-, and huge-model programs), **_heapset** maps to **_fheapset.** In small data models (tiny-, small-, and medium-model programs), **_heapset** maps to **_nheapset.**

Return Value

All four routines return an **int** whose value is one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	Initial header information cannot be found, or it is invalid
_HEAPBADNODE	Bad node has been found, or the heap is damaged
_HEAPEMPTY	Heap has not been initialized
_HEAPOK	Heap appears to be consistent

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

_heapchk functions, **_heapwalk** functions

Example

```
/* HEAPSET.C: This program checks the heap and fills in free entries
 * with the character 'Z'.
 */

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

void main()
{
   int heapstatus;
   char *buffer;
```

```
   if( (buffer = malloc( 1 )) == NULL )   /* Make sure heap is initialized */
      exit( 0 );
   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;
   }
   free( buffer );
}
```

Output

```
OK - heap is fine
```

Description

Traverse the heap and return information about the next entry.

include <malloc.h>

int _heapwalk(_HEAPINFO **entryinfo* **);**

int _bheapwalk(_segment *seg*, **_HEAPINFO** **entryinfo* **);**

int _fheapwalk(_HEAPINFO **entryinfo* **);**

int _nheapwalk(_HEAPINFO **entryinfo*);

entryinfo	Buffer to contain heap information
seg	Based-heap segment selector

Remarks

The **_heapwalk** family of routines helps debug heap-related problems in programs.

The **_heapwalk** routines walk through the heap, traversing one entry per call, and return a pointer to a **_heapinfo** structure that contains information about the next heap entry. The **_heapinfo** structure, defined in MALLOC.H, contains the following elements:

Element	Description
int far *_pentry	Heap entry pointer
size_t _size	Size of heap entry
int _useflag	Entry "in use" flag

A call to **_heapwalk** that returns **_HEAPOK** stores the size of the entry in the **_size** field and sets the **_useflag** field to either **_FREEENTRY** or **_USEDENTRY** (both are constants defined in MALLOC.H). To obtain this information about the first entry in the heap, pass the **_heapwalk** routine a pointer to a **_heapinfo** structure whose **_pentry** field is **NULL**.

The various **_heapwalk** functions walk through and gather information on these heaps:

Function	Heap Walked
_heapwalk	Depends on data model of program
_bheapwalk	Based heap specified by *seg* value; **_NULLSEG** specifies all based heaps
_fheapwalk	Far heap (outside default data segment)
_nheapwalk	Near heap (inside default data segment)

In large data models (that is, compact-, large-, and huge-model programs), **_heapwalk** maps to **_fheapwalk.** In small data models (tiny-, small-, and medium-model programs), **_heapwalk** maps to **_nheapwalk.**

Return Value

All three routines return one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	The initial header information cannot be found, or it is invalid.
_HEAPBADNODE	A bad node has been found, or the heap is damaged.
_HEAPBADPTR	The **_pentry** field of the **_heapinfo** structure does not contain a valid pointer into the heap.
_HEAPEND	The end of the heap has been reached successfully.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	No errors so far; the **_heapinfo** structure contains information about the next entry.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

_heapchk functions, **_heapset** functions

Example

```
/* HEAPWALK.C: This program "walks" the heap, starting at the beginning
 * (_pentry = NULL). It prints out each heap entry's use, location,
 * and size. It also prints out information about the overall state
 * of the heap as soon as _heapwalk returns a value other than _HEAPOK.
 */

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

```
void heapdump( void );

void main()
{
   char *buffer;

   heapdump();
   if( (buffer = malloc( 59 )) != NULL )
   {
      heapdump();
      free( buffer );
   }
   heapdump();
}

void heapdump( void )
{
   struct _heapinfo hinfo;
   int heapstatus;

   hinfo._pentry = NULL;
   while( ( heapstatus = _heapwalk( &hinfo ) ) == _HEAPOK )
   {
      printf( "%6s block at %Fp of size %4.4X\n",
         ( hinfo._useflag == _USEDENTRY ? "USED" : "FREE" ),
         hinfo._pentry, hinfo._size );
   }

   switch( heapstatus )
   {
      case _HEAPEMPTY:
         printf( "OK - empty heap\n" );
         break;
      case _HEAPEND:
         printf( "OK - end of heap\n" );
         break;
      case _HEAPBADPTR:
         printf( "ERROR - bad pointer to heap\n" );
         break;
      case _HEAPBADBEGIN:
         printf( "ERROR - bad start of heap\n" );
         break;
      case _HEAPBADNODE:
         printf( "ERROR - bad node in heap\n" );
         break;
   }
}
```

Output

```
  USED block at 0067:103E of size 000E
  USED block at 0067:104E of size 01F4
  USED block at 0067:1244 of size 0026
  USED block at 0067:126C of size 0200
  FREE block at 0067:146E of size 0B90
OK - end of heap
  USED block at 0067:103E of size 000E
  USED block at 0067:104E of size 01F4
  USED block at 0067:1244 of size 0026
  USED block at 0067:126C of size 0200
  USED block at 0067:146E of size 003C
  FREE block at 0067:14AC of size 0B52
OK - end of heap
  USED block at 0067:103E of size 000E
  USED block at 0067:104E of size 01F4
  USED block at 0067:1244 of size 0026
  USED block at 0067:126C of size 0200
  FREE block at 0067:146E of size 003C
  FREE block at 0067:14AC of size 0B52
OK - end of heap
```

Description

Frees a huge memory block.

#include <malloc.h> Required only for function declarations

void hfree(void _huge ****memblock* **);

memblock Pointer to allocated memory block

Remarks

The **hfree** function deallocates a memory block; the freed memory is returned to the operating system. The *memblock* 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.

Note that attempting to free an invalid *memblock* argument (one not allocated with **halloc**) may affect subsequent allocation and cause errors.

Return Value

None.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

halloc

Example

```
/* HALLOC.C: This program uses halloc to allocate space for 30,000 long
 * integers, then uses hfree to deallocate the memory.
 */

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

void main()
{
   long _huge *hbuf;

   /* Allocate huge buffer */
   hbuf = (long _huge *)halloc( 30000L, sizeof( long ) );
   if ( hbuf == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory successfully allocated\n" );
```

```
   /* Free huge buffer */
   hfree( hbuf );
}
```

Output

```
Memory successfully allocated
```

Description Calculate the hypotenuse.

#include <math.h>

double hypot(double *x*, **double** *y* **);**

long double hypotl(long double *x*, **long double** *y* **);**

x, *y*	Floating-point values

Remarks The **hypot** and **hypotl** functions calculate the length of the hypotenuse of a right triangle, given the length of the two sides *x* and *y* (or *xl* and *yl*). A call to **hypot** is equivalent to the following:

sqrt(*x* * *x* + *y* * *y***);**

The **hypotl** function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value The functions return the length of the hypotenuse. If an overflow results, the functions return **HUGE_VAL** and set **errno** to **ERANGE.**

Compatibility

hypot

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

hypotl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **cabs**

Example

```
/* HYPOT.C: This program prints the hypotenuse of a right triangle. */

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

```
void main()
{
   double x = 3.0, y = 4.0;

   printf( "If a right triangle has sides %2.1f and %2.1f, "
           "its hypotenuse is %2.1f\n", x, y, hypot( x, y ) );
}
```

Output

```
If a right triangle has sides 3.0 and 4.0, its hypotenuse is 5.0
```

Description

Get amount of memory required to store graphics images.

#include <graph.h>

long _far _imagesize(short *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

long _far _imagesize_w(double *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

long _far _imagesize_wxy(struct _wxycoord _far **pwxy1*,
struct _wxycoord _far **pwxy2* **);**

x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle

Remarks

The functions in the **_imagesize** family return the number of bytes needed to store the image defined by the bounding rectangle and specified by the coordinates given in the function call.

The **_imagesize** function defines the bounding rectangle in terms of view-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

The **_imagesize_w** function defines the bounding rectangle in terms of window-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

The **_imagesize_wxy** function defines the bounding rectangle in terms of the window-coordinate pairs *pwxy1* and *pwxy2*.

The number of bytes needed to store the image 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);
```

A call to **_getvideoconfig** stores the `bits_per_pixel` information in the **bitsperpixel** field of a **videoconfig** structure.

Return Value

The function returns the storage size of the image in bytes. There is no error return.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getimage** functions, **_getvideoconfig**, **_putimage** functions

Example See the example for **_getimage**.

Description

Input a byte (**inp**) or a word (**inpw**) from a port.

#include <conio.h> Required only for function declarations

int inp(unsigned *port* **);**

unsigned inpw(unsigned *port* **);**

port Port number

Remarks

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.

To use **inp** and **inpw** in OS/2 protected mode, you must use a .DEF file to declare the IOSEG segment that the run-time library uses to perform input/output on the port. In addition, the intrinsic (/Oi) versions of these functions do not work unless you put the code in a segment that is marked with the **IOPL** keyword in the .DEF file.

Because you cannot do IOPL from a regular code segment, the run-time library declares a separate code segment called **_IOSEG.** In order to use **inp, inpw, outp,** or **outpw** in any of the protected-mode run-time libraries (?LIBCP, LLIBCDLL, LLIBCMT, or CDLLOBJS-based DLL), you must have a .DEF file containing this line:

```
SEGMENTS _IOSEG CLASS 'IOSEG_CODE' IOPL
```

Return Value

The functions return the byte or word read from *port*. There is no error return.

Compatibility

□ ANSI ■ DOS ■ OS2 □ UNIX □ XENIX

See Also

outp, outpw

Example

See the example for **outp**.

Description

Executes the 8086 interrupt.

#include <dos.h>

int int86(int *intnum*, union REGS **inregs*, union REGS **outregs*);

intnum	Interrupt number
inregs	Register values on call
outregs	Register values on return

Remarks

The **int86** function executes the 8086-processor-family interrupt specified by the interrupt number *intnum*. 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 the *outregs* argument. The *inregs* and *outregs* arguments are unions of type **REGS**. The union type is defined in the include file DOS.H.

Do not use the **int86** function to call interrupts that modify the DS register. Instead, use the **int86x** function. The **int86x** function loads the DS and ES registers from the *segregs* parameter and also stores the DS and ES registers into *segregs* after the function call.

The **REGS** type is defined in the include file DOS.H.

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

bdos, int86x, intdos, intdosx

Example

```
/* INT86.C: This program uses int86 to call the BIOS video service
 * (INT 10H) to get information about the cursor.
 */

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

```
void main()
{
   union REGS inregs, outregs;

   /* Set up to get cursor information. */
   inregs.h.ah = 3;       /* Get Cursor Position function */
   inregs.h.bh = 0;       /* Page 0 */

   /* Execute video interrupt: */
   int86( 0x10, &inregs, &outregs );

   /* Display results. */
   printf( "Cursor position\n\tRow: %d\n\tColumn: %d\n",
           outregs.h.dh, outregs.h.dl );
   printf( "Cursor shape\n\tStart: %d\n\tEnd: %d\n",
           outregs.h.ch, outregs.h.cl );
}
```

Output

```
Cursor position
        Row: 2
        Column: 0
Cursor shape
        Start: 6
        End: 7
```

Description

Executes the 8086 interrupt; accepts segment-register values.

#include <dos.h>

int int86x(int *intnum*, **union REGS** **inregs*, **union REGS** **outregs*,
struct SREGS **segregs* **);**

intnum	Interrupt number
inregs	Register values on call
outregs	Register values on return
segregs	Segment-register values on call

Remarks

The **int86x** function executes the 8086-processor-family interrupt specified by the interrupt number *intnum*. Unlike the **int86** function, **int86x** accepts segment-register values in *segregs*, enabling programs that use large-model data segments or far pointers to 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 **REGS** and **SREGS** types are defined in the include file DOS.H.

Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

bdos, FP_SEG, int86, intdos, intdosx, segread

Example

```
/* INT86X.C: In this program, int86x executes an INT 21H instruction
 * to invoke 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.
 */

#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

char _far *filename = "int86x.c";

void main()
{
   union  REGS inregs, outregs;
   struct SREGS segregs;
   int    result;

   inregs.h.ah = 0x43;      /* DOS function to change attributes    */
   inregs.h.al = 0;         /* Subfunction 0 to get attributes)     */
   inregs.x.dx = FP_OFF( filename );   /* DS:DX points to file name */
   segregs.ds  = FP_SEG( filename );
   result = int86x( 0x21, &inregs, &outregs, &segregs );
   if( outregs.x.cflag )
      printf( "Can't get file attributes; error no. %d\n", result);
   else
      printf( "Attribs = 0x%.4x\n", outregs.x.cx );
}
```

Output

```
Attribs = 0x0020
```

Description

Executes the DOS system call.

#include <dos.h>

int intdos(union REGS **inregs*, **union REGS** **outregs* **);**

inregs	Register values on call
outregs	Register values on return

Remarks

The **intdos** function invokes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The **REGS** 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*. A nonzero **cflag** field indicates the flag was set by the system call and also indicates an error condition.

The **intdos** function is used to invoke DOS system calls that take arguments for input or output in registers other than DX (DH/DL) and AL. The **intdos** function is also used to invoke system calls that indicate errors by setting the carry flag. Under any other conditions, the **bdos** function can be used.

Do not use the **intdos** function to call interrupts that modify the DS register. Instead, use the **intdosx** or **int86x** function.

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

bdos, intdosx

Example

```
/* INTDOS.C: This program uses intdos to invoke DOS system call 2AH
 * (gets the current date).
 */

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

```
void main()
{
   union REGS inregs, outregs;

   inregs.h.ah = 0x2a;           /* DOS Get Date function: */
   intdos( &inregs, &outregs );
   printf( "Date: %d/%d/%d\n", outregs.h.dh, outregs.h.dl, outregs.x.cx );
}
```

Output

```
Date: 6/16/1989
```

Description

Executes the DOS system call; accepts segment-register values.

#include <dos.h>

int intdosx(union REGS ****inregs*, union REGS ****outregs***, struct SREGS ****segregs* **);**

inregs	Register values on call
outregs	Register values on return
segregs	Segment-register values on call

Remarks

The **intdosx** function invokes the DOS system call specified by register values defined in *inregs* and returns the results of the system call in *outregs*. Unlike the **intdos** function, **intdosx** accepts segment-register values in *segregs*, enabling programs that use large-model data segments or far pointers to specify which segment or pointer should be used during the system call. The **REGS** and **SREGS** 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*. A nonzero **cflag** field indicates the flag was set by the system call and also indicates an error condition.

The **intdosx** function is used to invoke DOS system calls that take an argument in the ES register or that take a DS register value different from the default data segment.

Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

bdos, FP_SEG, intdos, segread

Example

```
/* INTDOSX.C */
#include <dos.h>
#include <stdio.h>
```

```
char _far *buffer = "Dollar-sign terminated string\n\r\n\r$";

void main()
{
   union  REGS inregs, outregs;
   struct SREGS segregs;

   /* Print a $-terminated string on the screen using DOS function 0x09. */
   inregs.h.ah = 0x9;
   inregs.x.dx = FP_OFF( buffer );
   segregs.ds  = FP_SEG( buffer );
   intdosx( &inregs, &outregs, &segregs );
}
```

Output

```
Dollar-sign terminated string
```

Description

Test characters for specified conditions.

#include <ctype.h>

int isalnum(int *c* **);**

int isalpha(int *c* **);**

int isascii(int *c* **);**

int iscntrl(int *c* **);**

int isdigit(int *c* **);**

int isgraph(int *c* **);**

int islower(int *c* **);**

int isprint(int *c* **);**

int ispunct(int *c* **);**

int isspace(int *c* **);**

int isupper(int *c* **);**

int isxdigit(int *c* **);**

c	Integer to be tested

Remarks

Each function in the **is** family tests a given integer value, returning a nonzero value if the integer satisfies the test condition and 0 if it does not. The ASCII character set is assumed.

The **is** functions and their test conditions are listed below:

Function	Test Condition
isalnum	Alphanumeric ('A'–'Z', 'a'–'z', or '0'–'9')
isalpha	Letter ('A'–'Z' or 'a'–'z')
isascii	ASCII character (0x00 – 0x7F)
iscntrl	Control character (0x00 – 0x1F or 0x7F)
isdigit	Digit ('0'–'9')
isgraph	Printable character except space (' ')
islower	Lowercase letter ('a'–'z')

isprint	Printable character (0x20 – 0x7E)
ispunct	Punctuation character
isspace	White-space character (0x09 – 0x0D or 0x20)
isupper	Uppercase letter ('A'–'Z')
isxdigit	Hexadecimal digit ('A'–'F','a'–'f', or '0'–'9')

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).

These routines are implemented both as functions and as macros. For details on choosing a function or a macro implementation, see Section 1.4, "Choosing Between Functions and Macros."

Return Value

These routines return a nonzero value if the integer satisfies the test condition and 0 if it does not.

Compatibility

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

isascii

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

toascii, tolower, toupper functions

Example

```
/* ISFAM.C: This program tests all characters between 0x0 and 0x7F,
 * then displays each character with abbreviations for the character-type
 * codes that apply.
 */

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

```
void main()
{
   int ch;
   for( ch = 0; ch <= 0x7F; ch++ )
   {
      printf( "%.2x ", ch );
      printf( " %c", isprint( ch )  ? ch   : '\0' );
      printf( "%4s", isalnum( ch )  ? "AN" : "" );
      printf( "%3s", isalpha( ch )  ? "A"  : "" );
      printf( "%3s", isascii( ch )  ? "AS" : "" );
      printf( "%3s", iscntrl( ch )  ? "C"  : "" );
      printf( "%3s", isdigit( ch )  ? "D"  : "" );
      printf( "%3s", isgraph( ch )  ? "G"  : "" );
      printf( "%3s", islower( ch )  ? "L"  : "" );
      printf( "%3s", ispunct( ch )  ? "PU" : "" );
      printf( "%3s", isspace( ch )  ? "S"  : "" );
      printf( "%3s", isprint( ch )  ? "PR" : "" );
      printf( "%3s", isupper( ch )  ? "U"  : "" );
      printf( "%3s", isxdigit( ch ) ? "X"  : "" );
      printf( "\n" );
   }
}
```

Output

```
00          AS  C
01          AS  C
02          AS  C
.
.
.
38  8  AN     AS      D  G          PR     X
39  9  AN     AS      D  G          PR     X
3a  :         AS         G    PU    PR
3b  ;         AS         G    PU    PR
3c  <         AS         G    PU    PR
3d  =         AS         G    PU    PR
3e  >         AS         G    PU    PR
3f  ?         AS         G    PU    PR
40  @         AS         G    PU    PR
41  A  AN   A AS         G          PR  U  X
42  B  AN   A AS         G          PR  U  X
.
.
.
```

Description

Checks for a character device.

#include <io.h> Required only for function declarations

int isatty(int *handle* **);**

handle Handle referring to device to be tested

Remarks

The **isatty** function determines whether *handle* is associated with a character device (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.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

Example

```
/* ISATTY.C: This program checks to see whether stdout has been
 * redirected to a file.
 */

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

void main()
{
   if( isatty( fileno( stdout ) ) )
      printf( "stdout has not been redirected to a file\n" );
   else
      printf( "stdout has been redirected to a file\n");
}
```

Output

```
stdout has not been redirected to a file
```

Description

Converts an integer to a string.

#include <stdlib.h>	Required only for function declarations

char *itoa(int *value*, char **string*, int *radix*);

value	Number to be converted
string	String result
radix	Base of *value*

Remarks

The **itoa** function converts the digits of the given *value* argument 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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

ltoa, ultoa

Example

```
/* ITOA.C: This program converts integers of various sizes to strings
 * in various radixes.
 */

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

void main()
{
   char buffer[20];
   int  i = 3445;
   long l = -344115L;
   unsigned long ul = 1234567890UL;
```

```
   itoa( i, buffer, 10 );
   printf( "String of integer %d (radix 10): %s\n", i, buffer );
   itoa( i, buffer, 16 );
   printf( "String of integer %d (radix 16): 0x%s\n", i, buffer );
   itoa( i, buffer, 2  );
   printf( "String of integer %d (radix 2): %s\n", i, buffer );

   ltoa( l, buffer, 16 );
   printf( "String of long int %ld (radix 16): 0x%s\n", l, buffer );

   ultoa( ul, buffer, 16 );
   printf( "String of unsigned long %lu (radix 16): 0x%s\n", ul, buffer );
}
```

Output

```
String of integer 3445 (radix 10): 3445
String of integer 3445 (radix 16): 0xd75
String of integer 3445 (radix 2): 110101110101
String of long int -344115 (radix 16): 0xfffabfcd
String of unsigned long 1234567890 (radix 16): 0x499602d2
```

Description

Checks the console for keyboard input.

#include <conio.h> Required only for function declarations

int kbhit(void);

Remarks

The **kbhit** function checks the console for a recent keystroke. If the function returns a nonzero value, a keystroke is waiting in the buffer. The program can then call **getch** or **getche** to get the keystroke.

Return Value

The **kbhit** function returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* KBHIT.C: 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 get the keystroke.
 */

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

void main()
{
   /* Display message until key is pressed. */
   while( !kbhit() )
      cputs( "Hit me!! " );

   /* Use getch to throw key away. */
   printf( "\nKey struck was '%c'\n", getch() );
   getch();
}
```

Output

```
Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!!
Key struck was 'k'
```

Description Calculates the absolute value of a long integer.

#include <stdlib.h>	Required only for function declarations
#include <math.h>	

long labs(long *n* **);**

n	Long-integer value

Remarks 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **abs, cabs, fabs**

Example

```
/* ABS.C: This program computes and displays the absolute values of
 * several numbers.
 */

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

void main()
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;

   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);

   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);

   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

Output

```
The absolute value of -4 is 4
The absolute value of -41567 is 41567
The absolute value of -3.141593 is 3.141593
```

Description

Compute a real number from the mantissa and exponent.

#include <math.h>

double ldexp(double *x*, **int** *exp* **);**

long double ldexpl(long double *x*, **int** *exp* **);**

x	Floating-point value
exp	Integer exponent

Remarks

The **ldexp** and **ldexpl** functions calculate the value of $x * 2^{exp}$.

Return Value

The **ldexp** and **ldexpl** functions return $x * 2^{exp}$. If an overflow results, the functions return ± **HUGE_VAL** (depending on the sign of *x*) and set **errno** to **ERANGE**.

The **ldexpl** function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Compatibility

ldexp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

ldexpl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

frexp, modf

Example

```
/* LDEXP.C */
#include <math.h>
#include <stdio.h>
```

```
void main()
{
   double x = 4.0, y;
   int p = 3;

   y = ldexp( x, p );
   printf( "%2.1f times two to the power of %d is %2.1f\n", x, p, y );
}
```

Output

```
4.0 times two to the power of 3 is 32.0
```

Description

Computes the quotient and remainder of a long integer.

#include <stdlib.h>

ldiv_t ldiv (long int *numer*, **long int** *denom* **);**

numer	Numerator
denom	Denominator

Remarks

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 that 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, with the difference being that the arguments and the members of the returned structure are all of type **long int**.

The **ldiv_t** structure, defined in STDLIB.H, contains the following elements:

Element	Description
long int quot	Quotient
long int rem	Remainder

Return Value

The **ldiv** function returns a structure of type **ldiv_t**, comprising both the quotient and the remainder.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

div

Example

```
/* LDIV.C: This program takes two long integers as command-line
 * arguments and displays the results of the integer division.
 */

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

```
void main()
{
   long x = 5149627, y = 234879;
   ldiv_t div_result;

   div_result = ldiv( x, y );
   printf( "For %ld / %ld, the quotient is %ld, and the remainder is %ld\n",
           x, y, div_result.quot, div_result.rem );
}
```

Output

```
For 5149627 / 234879, the quotient is 21, and the remainder is 217168
```

Description

Performs a linear search for the specified key.

#include <search.h>	Required only for function declarations

char *lfind(const void ****key*, const void ****base***, unsigned int ****num***, unsigned int** ***width***,**
int (****compare* **)(const void ****elem1***, const void ****elem2*** **));**

key	Object to search for
base	Pointer to base of search data
num	Number of array elements
width	Width of array elements
compare( )	Pointer to comparison routine
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

Remarks

The **lfind** function performs a linear search for the value *key* in an array of *num* elements; each element is *width* bytes in size. (Unlike **bsearch**, **lfind** does not require the array to be sorted.) The *base* argument is a pointer to the base of the array to be searched.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. The **lfind** function calls 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	Elements are different
0	Elements are identical

Return Value

If the key is found, **lfind** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **lfind** returns **NULL**.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **bsearch, lsearch, qsort**

Example

```
/* LFIND.C: This program uses lfind to search for the word "hello"
 * in the command-line arguments.
 */

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

int compare( char **arg1, char **arg2 );

void main( int argc, char **argv )
{
   char **result;
   char *key = "hello";

   result = (char **)lfind( (char *)&key, (char *)argv,
                            &argc, sizeof( char * ), compare );
   if( result )
      printf( "%s found\n", *result );
   else
      printf( "hello not found!\n" );
}

int compare(char ** arg1, char **arg2 )
{
   return( strcmpi( *arg1, *arg2 ) );
}
```

Output

```
[C:\LIBREF] lfind What if I said Hello world
Hello found
```

Description Draw lines to specified points.

#include <graph.h>

short _far _lineto(short *x*, **short** *y* **);**

short _far _lineto_w(double *wx*, **double** *wy* **);**

x, *y*	End point
wx, *wy*	End point

Remarks The functions in the **_lineto** family draw a line from the current graphics position up to and including the destination point. The destination point for the **_lineto** function is given by the view-coordinate point (*x*, *y*). The destination point for the **_lineto_w** function is given by the window-coordinate point (*wx*, *wy*).

The line is drawn using the current color, logical write mode, and line style. If no error occurs, **_lineto** sets the current graphics position to the view-coordinate point (*x*, *y*); **_lineto_w** sets the current position to the window-coordinate point (*wx*, *wy*).

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** and **_lineto_w** routines return a nonzero value if anything is drawn; otherwise, they return 0.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getcurrentposition** functions, **_moveto** functions, **_setlinestyle**

Example

```
/* MOVETO.C: This program draws line segments of different colors. */

#include <graph.h>
#include <stdlib.h>
#include <conio.h>
```

```
void main()
{
   short x, y, xinc, yinc, color = 1;
   struct videoconfig v;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &v );
   xinc = v.numxpixels / 50;
   yinc = v.numypixels / 50;

   for( x = 0, y = v.numypixels - 1; x < v.numxpixels; x += xinc, y -= yinc )
   {
      _setcolor( color++ % 16 );
      _moveto( x, 0 );
      _lineto( 0, y );
   }
   getch();

   _setvideomode( _DEFAULTMODE );
}
```

Description

Gets detailed information on locale settings.

#include <locale.h>

struct lconv *localeconv(void);

Remarks

The **localeconv** function gets detailed information on the locale-specific settings for numeric formatting of the program's current locale. This information is stored in a structure of type **lconv**.

The **lconv** structure, defined in LOCALE.H, contains the following elements:

Element	Description
char *decimal_point	Decimal-point character for nonmonetary quantities.
char *thousands_sep	Character used to separate groups of digits to the left of the decimal point for non-monetary quantities.
char *grouping	Size of each group of digits in non-monetary quantities.
char *int_curr_symbol	International currency symbol for the current locale. The first three characters specify the alphabetic international currency symbol as defined in the *ISO 4217 Codes for the Representation of Currency and Funds* standard. The fourth character (immediately preceding the null character) is used to separate the international currency symbol from the monetary quantity.
char *currency_symbol	Local currency symbol for the current locale.
char *mon_decimal_point	Decimal-point character for monetary quantities.
char *mon_thousands_sep	Separator for groups of digits to the left of the decimal place in monetary quantities.
char *mon_grouping	Size of each group of digits in monetary quantities.
char *positive_sign	String denoting sign for nonnegative monetary quantities.

char *negative_sign	String denoting sign for negative monetary quantities.
char int_frac_digits	Number of digits to the right of the decimal point in internationally formatted monetary quantities.
char frac_digits	Number of digits to the right of the decimal point in formatted monetary quantities.
char p_cs_precedes	Set to 1 if the currency symbol precedes the value for a nonnegative formatted monetary quantity. Set to 0 if the symbol follows the value.
char p_sep_by_space	Set to 1 if the currency symbol is separated by a space from the value for a nonnegative formatted monetary quantity. Set to 0 if there is no space separation.
char n_cs_precedes	Set to 1 if the currency symbol precedes the value for a negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.
char n_sep_by_space	Set to 1 if the currency symbol is separated by a space from the value for a negative formatted monetary quantity. Set to 0 if there is no space separation.
char p_sign_posn	Position of positive sign in nonnegative formatted monetary quantities.
char n_sign_posn	Position of positive sign in negative formatted monetary quantities.

The elements of **grouping** and **mon_grouping** are interpreted according to the following rules:

Value	Interpretation
CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
n	The integer value *n* is the number of digits that make up the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values for **p_sign_posn** and **n_sign_posn** are interpreted according to the following rules:

Value	Interpretation
0	Parentheses surround the quantity and currency symbol
1	Sign string precedes the quantity and currency symbol
2	Sign string follows the quantity and currency symbol
3	Sign string immediately precedes the currency symbol
4	Sign string immediately follows the currency symbol

Return Value

The **localeconv** function returns a pointer to a structure of **lconv** type. Calls to the **setlocale** function with *category* values of **LC_ALL, LC_MONETARY,** or **LC_NUMERIC** will overwrite the contents of the structure.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

setlocale, strcoll, strftime, strxfrm

Description

Converts a time value and corrects for the local time zone.

#include <time.h>

struct tm *localtime(const time_t ***_timer_ **);

timer	Pointer to stored time

Remarks

The **localtime** function converts a time stored as a **time_t** value and stores the result in a structure of type **tm**. The **long** value *timer* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time; this value is usually obtained from the **time** function.

The fields of the structure type **tm** store the following values:

Element	Value Stored
int tm_sec	Seconds
int tm_min	Minutes
int tm_hour	Hours (0–24)
int tm_mday	Day of month (1–31)
int tm_mon	Month (0–11; January = 0)
int tm_year	Year (current year minus 1900)
int tm_wday	Day of week (0–6; Sunday = 0)
int tm_yday	Day of year (0–365; January 1 = 0)
int tm_isdst	Nonzero if daylight saving time is in effect, otherwise 0

Note that the **gmtime, mktime,** and **localtime** functions use a single statically allocated **tm** structure for the conversion. Each call to one of these routines destroys the result of the previous call.

The **localtime** function makes corrections for the local time zone if the user first sets the environment variable TZ. When TZ is set, three other environment variables (**timezone, daylight,** and **tzname**) are automatically set as well. See **tzset** for a description of these variables.

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. DOS and OS/2 do not accommodate dates prior to 1980. If the value in *timer* represents a date prior to January 1, 1980, the function returns **NULL**.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

asctime, ctime, ftime, gmtime, time, tzset

Example

```
/* LOCALTIM.C: 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).
 */

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

void main()
{
   struct tm *newtime;
   char  am_pm[] = "AM";
   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. */
      strcpy( 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 );
}
```

Output

```
Fri Jun 16 06:27:02 AM
```

Description

Locks or unlocks bytes of a file.

#include <sys\locking.h>

#include <io.h>	Required only for function declarations

int locking(int *handle*, **int** *mode*, **long** *nbytes* **);**

handle	File handle
mode	File-locking mode
nbytes	Number of bytes to lock

Remarks

The **locking** function locks or unlocks *nbytes* bytes of the file specified by *handle*. Locking bytes in a file prevents access to those bytes by other processes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbytes* bytes. It is possible to lock bytes past the end of the file.

The *mode* argument 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, immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, returns an error.
LK_NBLCK	Locks the specified bytes. If bytes cannot be locked, returns an error.
LK_NBRLCK	Same as **LK_NBLCK**.
LK_RLCK	Same as **LK_LOCK**.
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.

When a region of a file is being unlocked, it must correspond to a region that was previously locked. The **locking** function does not merge adjacent regions; if two locked regions are adjacent, each region must be unlocked separately.

Regions should be locked only briefly and should be unlocked before closing a file or exiting the program.

The **locking** function should be used only under OS/2 or under DOS versions 3.0 and later; it has no effect under earlier versions of DOS. Also, file sharing must be loaded to use the **locking** function. Note that under DOS versions 3.0 and 3.1, the files locked by parent processes may become unlocked when child processes exit.

Return Value

The **locking** function returns 0 if 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.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

creat, open

Example

```
/* LOCKING.C: This program opens a file with sharing. It locks some
 * bytes before reading them, then unlocks them. Note that the program
 * works correctly only if the following conditions are met:
 *     - The file exists
 *     - The program is run under OS/2, under DOS 3.0 or later
 *       with file sharing installed (SHARE.COM or SHARE.EXE), or
 *       if a Microsoft Networks compatible network is running
 */

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

```
void main()
{
   int  fh, numread;
   long pos, result;
   char buffer[40];

   /* Quit if can't open file or DOS version doesn't support sharing. */
   fh = sopen( "locking.c", O_RDWR, SH_DENYNO, S_IREAD | S_IWRITE );
   if( (fh == -1) || (_osmajor < 3) )
      exit( 1 );

   /* Lock some bytes and read them. Then unlock. */
   if( locking( fh, LK_NBLCK, 30L ) != -1 )
   {
      printf( "No one can change these bytes while I'm reading them\n" );
      numread = read( fh, buffer, 30 );
      printf( "%d bytes read: %.30s\n", numread, buffer );
      locking( fh, LK_UNLCK, 30L );
      printf( "Now I'm done. Do what you will with them\n" );
   }
   else
      perror( "Locking failed\n" );

   close( fh );
}
```

Output

```
No one can change these bytes while I'm reading them
30 bytes read: /* LOCKING.C: This program ope
Now I'm done. Do what you will with them
```

Description Calculate logarithms.

#include <math.h>

double log(double *x* **);**

double log10(double *x* **);**

long double logl(long double *x* **);**

long double log10l(long double *x* **);**

x	Value whose logarithm is to be found

Remarks The **log** and **log10** functions calculate the natural logarithm and the base-10 logarithm, respectively, of *x*. The **logl** and **log10l** functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value The **log** functions return the logarithm of the argument *x*. If *x* is negative, the functions print a **DOMAIN** error message to **stderr**, return the value –**HUGE_VAL**, and set **errno** to **EDOM**. If *x* is 0, the functions print a **SING** error message to **stderr**, return the value –**HUGE_VAL**, and set **errno** to **ERANGE**.

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

Compatibility **log, log10**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

logl, log10l

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **exp, matherr, pow** functions

Example

```
/* LOG.C: This program uses log and log10 to calculate the natural
 * logarithm and the base-10 logarithm of 9,000.
 */

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

```
void main()
{
   double x = 9000.0;
   double y;

   y = log( x );
   printf( "log( %.2f ) = %f\n", x, y );
   y = log10( x );
   printf( "log10( %.2f ) = %f\n", x, y );
}
```

Output

```
log( 9000.00 ) = 9.104980
log10( 9000.00 ) = 3.954243
```

The 8087 family of numeric coprocessor chips supports the 80-bit precision floating-point data type. In Microsoft C, version 6.0, the long double functions, whose names end with **l**, map the C **long double** type into this 80-bit, 10-byte form. Unlike the regular floating-point functions (such as **acos**), which return values of type **double**, these long double functions (such as **acosl**) return values of type **long double**. The long double functions also return their values on the coprocessor stack for all calling conventions.

The long double type is also supported by the addition of the "L" prefix for a floating-point format specification in the **printf** and **scanf** family of functions.

The long double versions are described on the reference pages for their regular counterparts. These are the regular C run-time math functions with corresponding long double equivalents:

Regular Function	Long Double Form
acos	**acosl**
asin	**asinl**
atan	**atanl**
atan2	**atan2l**
atof	**_atold**
cabs	**cabsl**
ceil	**ceill**
cos	**cosl**
cosh	**coshl**
exp	**expl**
fabs	**fabsl**
floor	**floorl**
fmod	**fmodl**
frexp	**frexpl**
hypot	**hypotl**
ldexp	**ldexpl**
log	**logl**
log10	**log10l**
matherr	**_matherrl**
modf	**modfl**

pow	**powl**
sin	**sinl**
sinh	**sinhl**
sqrt	**sqrtl**
tan	**tanl**
tanh	**tanhl**

Description

Restores stack environment and execution locale.

#include <setjmp.h>

void longjmp(jmp_buf *env*, **int** *value* **);**

env	Variable in which environment is stored
value	Value to be returned to **setjmp** call

Remarks

The **longjmp** function restores a stack environment and execution locale previously saved in *env* by **setjmp**. The **setjmp** and **longjmp** functions 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 call and 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) that are 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 results.

The value returned by **setjmp** must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.

Observe the following three restrictions when using **longjmp**:

1. Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling **setjmp** may not be restored to the proper values after **longjmp** is executed.
2. Do not use **longjmp** to transfer control from within one overlay to within another. The overlay manager keeps the overlay in memory after a call to **longjmp**.
3. Do not use **longjmp** to transfer control out of an interrupt-handling routine unless the interrupt is caused by a floating-point exception. In this case, a program may return from an interrupt handler via **longjmp** if it first reinitializes the floating-point math package by calling **_fpreset**.

Return Value

None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **setjmp**

Example See the example for **_fpreset.**

Description Rotate bits to the left (**_lrotl**) or right (**_lrotr**).

#include <stdlib.h>

unsigned long _lrotl(unsigned long *value*, **int** *shift* **);**

unsigned long _lrotr(unsigned long *value*, **int** *shift* **);**

value	Value to be rotated
shift	Number of bits to shift

Remarks The **_lrotl** and **_lrotr** functions rotate *value* by *shift* bits. The **_lrotl** function rotates the value left. The **_lrotr** function rotates the value right. Both functions "wrap" bits rotated off one end of *value* to the other end.

Return Value Both functions return the rotated value. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_rotl**, **_rotr**

Example

```
/* LROT.C */
#include <stdlib.h>
#include <stdio.h>

void main()
{
   unsigned long val = 0x0fac35791;

   printf( "0x%8.8lx rotated left eight times is 0x%8.8lx\n",
           val, _lrotl( val, 8 ) );
   printf( "0x%8.8lx rotated right four times is 0x%8.8lx\n",
           val, _lrotr( val, 4 ) );
}
```

Output

```
0xfac35791 rotated left eight times is 0xc35791fa
0xfac35791 rotated right four times is 0x1fac3579
```

Description

Performs a linear search for a value; adds to end of list if not found.

#include <search.h>	Required only for function declarations

char *lsearch(const void ****key*, const void ****base***, unsigned int ****num***,**
unsigned int *width***, int (****compare* **)(const void ****elem1***, const void ****elem2* **));**

key	Object to search for
base	Pointer to base of search data
num	Number of elements
width	Width of elements
compare	Pointer to comparison routine
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

Remarks

The **lsearch** function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes in size. (Unlike **bsearch**, **lsearch** does not require the array to be sorted.) The *base* argument is a pointer to the base of the array to be searched.

If *key* is not found, **lsearch** adds it to the end of the array.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The **lsearch** function calls 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	Elements are different
0	Elements are identical

Return Value

If the key is found, **lsearch** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **lsearch** returns a pointer to the newly added item at the end of the array.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **bsearch, lfind**

Example See the example for **lfind**.

Description

Moves a file pointer to the specified location.

#include <io.h>	Required only for function declarations
#include <stdio.h>	

long lseek(int *handle*, **long** *offset*, **int** *origin* **);**

handle	Handle referring to open file
offset	Number of bytes from *origin*
origin	Initial position

Remarks

The **lseek** function moves the file pointer 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* argument must be one of the following constants, which are 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. The function returns –1L to indicate an error and sets **errno** 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.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fseek, tell**

Example

```
/* LSEEK.C: This program first opens a file named LSEEK.C.
 * 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.
 */

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


void main()
{
   int fh;
   long pos;               /* Position of file pointer */
   char buffer[10];

   fh = open( "lseek.c", O_RDONLY );

   /* Seek the beginning of the file: */
   pos = lseek( fh, 0L, SEEK_SET );
   if( pos == -1L )
      perror( "lseek to beginning failed" );
   else
      printf( "Position for beginning of file seek = %ld\n", pos );

   /* Move file pointer a little */
   read( fh, buffer, 10 );

   /* 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 = %ld\n", pos );
```

```
   /* Set the end of the file: */
   pos = lseek( fh, ØL, SEEK_END );
   if( pos == -1L )
      perror( "lseek to end failed" );
   else
      printf( "Position for end of file seek = %ld\n", pos );

   close( fh );
}
```

Output

```
Position for beginning of file seek = Ø
Position for current position seek = 1Ø
Position for end of file seek = 1183
```

Description

Converts a long integer to a string.

#include <stdlib.h> Required only for function declarations

char *ltoa(long *value*, **char** **string*, **int** *radix* **);**

value	Number to be converted
string	String result
radix	Base of *value*

Remarks

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*, which 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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

itoa, ultoa

Example

```
/* ITOA.C: This program converts integers of various sizes to strings
 * in various radixes.
 */

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

void main()
{
   char buffer[20];
   int  i = 3445;
   long l = -344115L;
   unsigned long ul = 1234567890UL;
```

```
   itoa( i, buffer, 10 );
   printf( "String of integer %d (radix 10): %s\n", i, buffer );
   itoa( i, buffer, 16 );
   printf( "String of integer %d (radix 16): 0x%s\n", i, buffer );
   itoa( i, buffer, 2  );
   printf( "String of integer %d (radix 2): %s\n", i, buffer );

   ltoa( l, buffer, 16 );
   printf( "String of long int %ld (radix 16): 0x%s\n", l, buffer );

   ultoa( ul, buffer, 16 );
   printf( "String of unsigned long %lu (radix 16): 0x%s\n", ul, buffer );
}
```

Output

```
String of integer 3445 (radix 10): 3445
String of integer 3445 (radix 16): 0xd75
String of integer 3445 (radix 2): 110101110101
String of long int -344115 (radix 16): 0xfffabfcd
String of unsigned long 1234567890 (radix 16): 0x499602d2
```

Description

Creates a single path name.

#include <stdlib.h>

void _makepath(char **path*, **char** **drive*, **char** **dir*, **char** **fname*, **char** **ext* **);**

path	Full path-name buffer
drive	Drive letter
dir	Directory path
fname	File name
ext	File extension

Remarks

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 the **_makepath** function can handle. The other arguments point to 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 *dir* 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. If *fname* is **NULL** or points to an empty string, no file name is inserted in the composite *path* string.

ext The *ext* argument contains the actual file-name extension, with or without a leading period (.). The **_makepath** routine will insert the period automatically if it does not 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 must be no larger than the **_MAX_PATH** constant. The **_MAX_PATH** limit permits a path name much larger than any of the current versions of DOS or OS/2 will handle.

Return Value None.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_fullpath**, **_splitpath**

Example

```
/* MAKEPATH.C */
#include <stdlib.h>
#include <stdio.h>

void main()
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath( path_buffer, "c", "\\c60\\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( "  Filename: %s\n", fname );
   printf( "  Ext: %s\n", ext );
}
```

Output

```
Path created with _makepath: c:\c60\clibref\makepath.c

Path extracted with _splitpath:
  Drive: c:
  Dir: \c60\clibref\
  Filename: makepath
  Ext: .c
```

Description

Allocate memory blocks.

#include <stdlib.h>	For ANSI compatibility (**malloc** only)
#include <malloc.h>	Required only for function declarations

void *malloc(size_t *size* **);**

void _based(void) *_bmalloc(_segment *seg*, **size_t** *size* **);**

void _far *_fmalloc(size_t *size* **);**

void _near *_nmalloc(size_t *size* **);**

size	Bytes to allocate
seg	Based heap segment selector

Remarks

Functions in the **malloc** family allocate a memory block of at least *size* bytes. The block may be larger than *size* bytes because of space required for alignment and maintenance information. If *size* is 0, each of these functions allocates a zero-length item in the heap and returns a valid pointer to that item.

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.

In large data models (compact-, large-, and huge-model programs), **malloc** maps to **_fmalloc**. In small data models (tiny-, small-, and medium-model programs), **malloc** maps to **_nmalloc**.

The **_fmalloc** function allocates a memory block of at least *size* bytes in the far heap, which is outside the default data segment. The return value is a far pointer to **void**.

The **_bmalloc** function allocates a memory block of at least *size* bytes in the based heap segment specified by the segment selector *seg*.

The **malloc** functions allocate memory in the heap segment specified below.

Function	Heap Segment
malloc	Depends on data model of program
_bmalloc	Based heap segment specified by *seg* value
_fmalloc	Far heap (outside default data segment)
_nmalloc	Near heap (within default data segment)

If you are creating programs to run in both real mode and protected mode, you should probably bind with APILMR.OBJ as well as API.LIB and OS2.LIB. This is necessary if a program will use the **_nmalloc** function.

The functions listed below call the **malloc** family of routines. In addition, the C start-up code uses **malloc** to allocate storage for the **environ/envp** and **argv** strings and arrays.

The following routines call **malloc**:

calloc	**fseek**	**_searchenv**
execv	**fsetpos**	**spawnv**
execve	**fullpath**	**spawnve**
execvp	**fwrite**	**spawnvp**
execvpe	**getc**	**spawnvpe**
execl	**getchar**	**spawnl**
execle	**getcwd**	**spawnle**
execlp	**_getcwd**	**spawnlp**
execlpe	**gets**	**spawnlpe**
fgetc	**getw**	**strdup**
fgetchar	**_popen**	**system**
fgets	**printf**	**scanf**
fprint	**putc**	**setvbuf**
fputc	**putchar**	**tempnam**
fputchar	**putenv**	**ungetc**
fputs	**puts**	**vfprintf**
fread	**putw**	**vprintf**
fscanf		

The following routines call **malloc** only in the multithread run-time libraries (LLIBCMT, LLIBCDLL, CDLLOBJS), not in the regular run-time libraries:

asctime	**localtime**	**_strerrpr**
_beginthread	**mktime**	**tmpfile**
ctime	**sterror**	**tmpnam**
gmtime		

The following routines call **_nmalloc**:

_nrealloc
_ncalloc
_nstrdup
realloc (in small data models)

The following routines call **_fmalloc**:

_frealloc
_fcalloc
_fstrdup
realloc (in large data models)

C5.1 Differences *In Microsoft C version 5.1, the* ***_fmalloc*** *function would retry allocating within the default data segment (i.e., in the near heap) if sufficient memory was not available outside the default data segment. Version 6.0 returns* ***NULL*** *under these conditions.*

In version 5.1, the start-up code used ***malloc*** *only if wild-card expansion was used.*

The ***_freect***, ***_memavl***, *and* ***_memmax*** *functions called* ***malloc*** *in version 5.1 but do not do so in version 6.0.*

Return Value

The **malloc** function returns a **void** pointer to the allocated space. The **_nmalloc** function returns a (**void _near** *) and **_fmalloc** returns a (**void _far** *). The **_bmalloc** function returns a (**void _based(void)** *).

The **_malloc**, **_fmalloc** and **_nmalloc** functions return **NULL** if there is insufficient memory available. The **_bmalloc** function returns **_NULLOFF** if there is insufficient memory available.

Always check the return from the **malloc** function, even if the amount of memory requested is small.

Compatibility

malloc

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_bmalloc, _fmalloc, _nmalloc

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

calloc functions, **free** functions, **realloc** functions

Example

```
/* MALLOC.C: This program allocates memory with malloc, then frees
 * the memory with free.
 */

#include <stdlib.h>         /* Definition of _MAX_PATH */
#include <stdio.h>
#include <malloc.h>

void main()
{
   char *string;

   /* Allocate space for a path name */
   string = malloc( _MAX_PATH );
   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory space allocated for pathname\n" );
   free( string );
   printf( "Memory freed\n" );
}
```

Output

```
Memory space allocated for pathname
Memory freed
```

Description Handle math errors.

#include <math.h>

int matherr(struct exception **except* **);**

int _matherrl(struct _exceptionl **except* **);**

except	Pointer to structure containing error information

Remarks The **matherr** functions process errors generated by the functions of the math library. The math functions call the appropriate **matherr** routine whenever an error is detected. The **_matherrl** function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

The user can provide a different definition of the **matherr** or **_matherrl** 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 **exception** structure contains the following elements:

Element	Description
int type	Exception type
char *name	Name of function where error occurred
double arg1, arg2	First and second (if any) argument to function
double retval	Value to be returned by function

The **type** specifies the type of math error. It is 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 the **matherr** function 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** functions should return 0 to indicate an error, and a nonzero value to indicate successful corrective action.

Compatibility

matherr

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_matherrl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

acos functions, **asin** functions, **atan** functions, **bessel** functions, **cabs**, **cos** functions, **exp**, **hypot**, **log** functions, **pow**, **sin** functions, **sqrt**, **tan** functions

Example

```
/* MATHERR.C: To use matherr, you must turn off the Extended Dictionary
 * flag within the Microsoft Programmer's WorkBench environment, or use the
 * /NOE linker option outside the environment. For example:
 *      CL matherr.c /link /NOE
 */

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

```
void main()
{
   /* Do several math operations that cause errors. The matherr
    * routine handles DOMAIN errors, but lets the system handle
    * other errors normally.
    */
   printf( "log( -2.0 ) = %e\n", log( -2.0 ) );
   printf( "log10( -5.0 ) = %e\n", log10( -5.0 ) );
   printf( "log( 0.0 ) = %e\n", log( 0.0 ) );
}

/* Handle several math errors caused by passing a negative argument
 * to log or log10 (DOMAIN errors). When this happens, matherr returns
 * the natural or base-10 logarithm of the absolute value of the
 * argument and suppresses the usual error message.
 */
int matherr( struct exception *except )
{
    /* Handle DOMAIN errors for log or log10. */
    if( except->type == DOMAIN )
    {
        if( strcmp( except->name, "log" ) == 0 )
        {
            except->retval = log( -(except->arg1) );
            printf( "Special: using absolute value: %s: DOMAIN error\n",
                    except->name );
            return 1;
        }
        else if( strcmp( except->name, "log10" ) == 0 )
        {
            except->retval = log10( -(except->arg1) );
            printf( "Special: using absolute value: %s: DOMAIN error\n",
                    except->name );
            return 1;
        }
    }
    else
    {
        printf( "Normal: " );
        return 0;    /* Else use the default actions */
    }
}
```

Output

```
Special: using absolute value: log: DOMAIN error
log( -2.0 ) = 6.931472e-001
Special: using absolute value: log10: DOMAIN error
log10( -5.0 ) = 6.989700e-001
Normal: log: SING error
log( 0.0 ) = -1.797693e+308
```

Description Returns the larger of two values.

#include <stdlib.h>

type **max(** *type a*, *type b* **);**

type	Any numeric data type
a, *b*	Values of any numeric type to be compared

Remarks The **max** macro compares two values and returns the value of the larger one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Return Value The macro returns the larger of the two arguments.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **min**

Example

```
/* MINMAX.C */
#include <stdlib.h>
#include <stdio.h>

void main()
{
   int a = 10;
   int b = 21;

   printf( "The larger of %d and %d is %d\n",  a, b, max( a, b ) );
   printf( "The smaller of %d and %d is %d\n", a, b, min( a, b ) );
}
```

Output

```
The larger of 10 and 21 is 21
The smaller of 10 and 21 is 10
```

Description

Returns the size of memory available.

#include <malloc.h> Required only for function declarations

size_t _memavl(void);

Remarks

The **_memavl** function returns the approximate size, in bytes, of the memory available for dynamic memory allocation in the near heap (default data segment). The **_memavl** function can be used with **calloc**, **malloc**, or **realloc** in tiny, small, and medium memory models and with **_ncalloc**, **_nmalloc** or **_nrealloc** in any memory model.

The number returned by the **_memavl** function may not be the number of contiguous bytes. Consequently, a call to **malloc** requesting allocation of the size returned by **_memavl** may not succeed. Use the **_memmax** function to find the size of the largest available contiguous block of memory.

Return Value

The **_memavl** function returns the size in bytes as an unsigned integer.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

calloc functions, **_freect, malloc** functions, **_memmax, realloc** functions

Example

```
/* MEMAVL.C: This program uses _memavl to determine the amount of
 * memory available for dynamic allocation. It then uses malloc to
 * allocate space for 5,000 long integers and uses _memavl again to
 * determine the new amount of available memory.
 */

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

void main()
{
   long *longptr;

   printf( "Memory available before _nmalloc = %u\n", _memavl() );
   if( (longptr = _nmalloc( 5000 * sizeof( long ) )) != NULL )
   {
      printf( "Memory available after _nmalloc = %u\n", _memavl() );
      _nfree( longptr );
   }
}
```

Output

```
Memory available before _nmalloc = 60906
Memory available after _nmalloc = 40390
```

Description

Copy characters from a buffer.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H or MEMORY.H

void *memccpy(void *dest, void *src, int c, unsigned int count);

void _far * _far _fmemccpy(void _far *dest, void _far *src, int c, unsigned int count);

dest	Pointer to destination
src	Pointer to source
c	Last character to copy
count	Number of characters

Remarks

The **memccpy** and **_fmemccpy** functions copy 0 or more bytes of *src* to *dest*, halting when the character *c* has been copied or when *count* bytes have been copied, whichever comes first.

The **_fmemccpy** function is a model-independent (large-model) form of the **memccpy** function. It can be called from any point in any program.

Return Value

If the character *c* is copied, **memccpy** or **_fmemccpy** returns a pointer (or far pointer) to the byte in *dest* that immediately follows the character. If *c* is not copied, **memccpy** returns **NULL**.

Compatibility

memccpy

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

_fmemccpy

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

memchr, memcmp, memcpy, memset

Example

```
/* MEMCCPY.C */
#include <memory.h>
#include <stdio.h>
#include <string.h>
```

```
char string1[60] = "The quick brown dog jumps over the lazy fox";

void main()
{
   char buffer[61];
   char *pdest;

   printf( "Function:\tmemccpy 60 characters or to character 's'\n" );
   printf( "Source:\t\t%s\n", string1 );
   pdest = memccpy( buffer, string1, 's', 60 );
   *pdest = '\0';
   printf( "Result:\t\t%s\n", buffer );
   printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );
}
```

Output

```
Function:       memccpy 60 characters or to character 's'
Source:         The quick brown dog jumps over the lazy fox
Result:         The quick brown dog jumps
Length:         25 characters
```

Description

Find characters in a buffer.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H (for ANSI compatibility) or MEMORY.H

void *memchr(const void ****buf*, int** *c***, size_t** *count* **);**

void _far * _far _fmemchr(const void _far ****buf*, int** *c***, size_t** *count* **);**

buf	Pointer to buffer
c	Character to look for
count	Number of characters

Remarks

The **memchr** and **_fmemchr** functions look for the first occurrence of *c* in the first *count* bytes of *buf*. They stop when they find *c* or when they have checked the first *count* bytes.

The **_fmemchr** function is a model-independent (large-model) form of the **memchr** function. It can be called from any point in any program.

Return Value

If successful, **memchr** or **_fmemchr** returns a pointer (or a far pointer) to the first location of *c* in *buf*. Otherwise, they return **NULL**.

Compatibility

memchr

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fmemchr

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

memccpy, memcmp, memcpy, memset, strchr

Example

```
/* MEMCHR.C */
#include <memory.h>
#include <stdio.h>
```

```
int  ch = 'r';
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

void main()
{
   char *pdest;
   int result;

   printf( "String to be searched:\n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );

   printf( "Search char:\t%c\n", ch );
   pdest = memchr( string, ch, sizeof( string ) );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\t\t%c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t\t%c not found\n" );
}
```

Output

```
String to be searched:
                The quick brown dog jumps over the lazy fox
                         1         2         3         4         5
                12345678901234567890123456789012345678901234567890

Search char:    r
Result:         r found at position 12
```

Description

Compare characters in two buffers.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H (for ANSI compatibility) or MEMORY.H

int memcmp(const void **buf1*, **const void** **buf2*, **size_t** *count* **);**

int _far _fmemcmp(const void _far **buf1*, **const void _far** **buf2*, **size_t** *count* **);**

buf1	First buffer
buf2	Second buffer
count	Number of characters

Remarks

The **memcmp** and **_fmemcmp** functions compare the first *count* bytes of *buf1* and *buf2* and return a value indicating their relationship, as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

The **_fmemcmp** function is a model-independent (large-model) form of the **memcmp** function. It can be called from any point in program.

There is a semantic difference between the function version of **memcmp** and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value

The **memcmp** function returns an integer value, as described above.

Compatibility

memcmp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fmemcmp

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also memccpy, memchr, memcpy, memset, strcmp, strncmp

Example

```
/* MEMCMP.C: This program uses memcmp to compare the strings named
 * first and second. If the first 19 bytes of the strings are
 * equal, the program considers the strings to be equal.
 */

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

void main()
{
   char first[]  = "12345678901234567890";
   char second[] = "12345678901234567891";
   int result;

   printf( "Compare '%.19s' to '%.19s':\n", first, second );
   result = memcmp( first, second, 19 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
   printf( "Compare '%.20s' to '%.20s':\n", first, second );
   result = memcmp( first, second, 20 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

Output

```
Compare '12345678901234567890' to '12345678901234567890':
First is equal to second.
Compare '12345678901234567890' to '12345678901234567891':
First is less than second.
```

Description Copy characters between buffers.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H (for ANSI compatibility) or MEMORY.H

void *memcpy(void ***dest*****, const void** ***src*****, size_t** *count* **);**

void _far * _far _fmemcpy(void _far ***dest*****, const void _far** ***src*****, size_t** *count* **);**

dest	New buffer
src	Buffer to copy from
count	Number of characters to copy

Remarks The **memcpy** and **_fmemcpy** functions copy *count* bytes of *src* to *dest*. If the source and destination overlap, these functions do not ensure that the original source bytes in the overlapping region are copied before being overwritten. Use **memmove** to handle overlapping regions.

The **_fmemcpy** function is a model-independent (large-model) form of the **memcpy** function. It can be called from any point in any program.

There is a semantic difference between the function version of **memcpy** and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value The **memcpy** and **_fmemcpy** functions return a pointer to *dest*.

Compatibility **memcpy**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fmemcpy

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also memccpy, memchr, memcmp, memmove, memset, strcpy, strncpy

Example

```
/* MEMCPY.C. Illustrate overlapping copy: memmove handles it
 * correctly; memcpy does not.
 */
#include <memory.h>
#include <string.h>
#include <stdio.h>

char string1[60] = "The quick brown dog jumps over the lazy fox";
char string2[60] = "The quick brown fox jumps over the lazy dog";
/*                          1         2         3         4         5
 *                 12345678901234567890123456789012345678901234567890
 */
void main()
{
   printf( "Function:\tmemcpy without overlap\n" );
   printf( "Source:\t\t%s\n", string1 + 40 );
   printf( "Destination:\t%s\n", string1 + 16 );
   memcpy( string1 + 16, string1 + 40, 3 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );

   /* Restore string1 to original contents */
   memcpy( string1 + 16, string2 + 40, 3 );

   printf( "Function:\tmemmove with overlap\n" );
   printf( "Source:\t\t%s\n", string2 + 4 );
   printf( "Destination:\t%s\n", string2 + 10 );
   memmove( string2 + 10, string2 + 4, 40 );
   printf( "Result:\t\t%s\n", string2 );
   printf( "Length:\t\t%d characters\n\n", strlen( string2 ) );

   printf( "Function:\tmemcpy with overlap\n" );
   printf( "Source:\t\t%s\n", string1 + 4 );
   printf( "Destination:\t%s\n", string1 + 10 );
   memcpy( string1 + 10, string1 + 4, 40 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );

}
```

Output

```
Function:       memcpy without overlap
Source:         fox
Destination:    dog jumps over the lazy fox
Result:         The quick brown fox jumps over the lazy fox
Length:         43 characters

Function:       memmove with overlap
Source:         quick brown fox jumps over the lazy dog
Destination:    brown fox jumps over the lazy dog
Result:         The quick quick brown fox jumps over the lazy dog
Length:         49 characters

Function:       memcpy with overlap
Source:         quick brown dog jumps over the lazy fox
Destination:    brown dog jumps over the lazy fox
Result:         The quick quick quick quick quick quick quick quic
Length:         50 characters
```

Description

Compare characters in two buffers (case-insensitive).

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H or MEMORY.H

int memicmp(void **buf1*, **void** **buf2*, **unsigned int** *count* **);**

int _far _fmemicmp(void _far **buf1*, **void _far** **buf2*, **unsigned int** *count* **);**

buf1	First buffer
buf2	Second buffer
count	Number of characters

Remarks

The **memicmp** and **_fmemicmp** functions compare the first *count* characters of the two buffers *buf1* and *buf2* byte-by-byte. The comparison is made without regard to the case of letters in the two buffers; that is, uppercase and lowercase letters are considered equivalent. The **memicmp** and **_fmemicmp** functions return a value indicating the relationship of the two buffers, as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

The **_fmemicmp** function is a model-independent (large-model) form of the **memicmp** function. It can be called from any point in any program.

Return Value

The **memicmp** and **_fmemicmp** functions return an integer value, as described above.

Compatibility

memicmp

☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fmemicmp

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also memccpy, memchr, memcmp, memcpy, memset, stricmp, strnicmp

Example

```
/* MEMICMP.C: This program uses memicmp to compare the first
 * 29 letters of the strings named first and second without
 * regard to the case of the letters.
 */

#include <memory.h>
#include <stdio.h>
#include <string.h>

void main()
{
   int result;
   char first[]  = "Those Who Will Not Learn from History";
   char second[] = "THOSE WHO WILL NOT LEARN FROM their mistakes";
   /* Note that the 29th character is right here ^ */

   printf( "Compare '%.29s' to '%.29s'\n", first, second );
   result = memicmp( first, second, 29 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

Output

```
Compare 'Those Who Will Not Learn from' to 'THOSE WHO WILL NOT LEARN FROM'
First is equal to second.
```

Description

Finds the size of the largest contiguous memory block.

#include <malloc.h>

size_t _memmax(void);

Remarks

The **_memmax** function returns the size (in bytes) of the largest contiguous block of memory that can be allocated from the near heap (i.e., the default data segment). Calling **_nmalloc** with the value returned by the **_memmax** function will succeed as long as **_memmax** returns a nonzero value.

Return Value

The function returns the block size, if successful. Otherwise, it returns 0, indicating that nothing more can be allocated from the near heap.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

malloc functions, **msize** functions

Example

```
/* MEMMAX.C: This program uses _memmax and _nmalloc to allocate
 * the largest block of memory available in the near heap.
 */

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

void main()
{
   size_t contig;
   char *p;
```

```
   /* Determine contiguous memory size */
   contig = _memmax();
   printf( "Largest block of available memory is %u bytes long\n", contig );
   if( contig )
   {
      p = _nmalloc( contig * sizeof( int ) );
      if( p == NULL )
         printf( "Error with malloc (should never occur)\n" );
      else
      {
         printf( "Maximum allocation succeeded\n" );
         free( p );
      }
   }
   else
      printf( "Near heap is already full\n" );
}
```

Output

```
Largest block of available memory is 60844 bytes long
Maximum allocation succeeded
```

Description

Move one buffer to another.

#include <string.h>

void *memmove(void ***dest*, const void ***src***, size_t** *count* **);**

void _far * _far _fmemmove(void _far ***dest*, const void _far ***src***, size_t** *count* **);**

dest	Destination object
src	Source object
count	Number of characters to copy

Remarks

The **memmove** and **_fmemmove** functions copy *count* characters from the source (*src*) to the destination (*dest*). If some regions of the source area and the destination overlap, the **memmove** and **_fmemmove** functions ensure that the original source bytes in the overlapping region are copied before being overwritten.

The **_fmemmove** function is a model-independent (large-model) form of the **memmove** function. It can be called from any point in any program.

Return Value

The **memmove** and **_fmemmove** functions return the value of *dest*.

Compatibility

memmove

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

_fmemmove

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

memccpy, memcpy, strccpy, strncpy

Example

```
/* MEMCPY.C. Illustrate overlapping copy: memmove handles it
 * correctly; memcpy does not.
 */
#include <memory.h>
#include <string.h>
#include <stdio.h>
```

```
char string1[60] = "The quick brown dog jumps over the lazy fox";
char string2[60] = "The quick brown fox jumps over the lazy dog";
/*                           1         2         3         4         5
 *                  12345678901234567890123456789012345678901234567890
 */
void main()
{
   printf( "Function:\tmemcpy without overlap\n" );
   printf( "Source:\t\t%s\n", string1 + 40 );
   printf( "Destination:\t%s\n", string1 + 16 );
   memcpy( string1 + 16, string1 + 40, 3 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );

   /* Restore string1 to original contents */
   memcpy( string1 + 16, string2 + 40, 3 );

   printf( "Function:\tmemmove with overlap\n" );
   printf( "Source:\t\t%s\n", string2 + 4 );
   printf( "Destination:\t%s\n", string2 + 10 );
   memmove( string2 + 10, string2 + 4, 40 );
   printf( "Result:\t\t%s\n", string2 );
   printf( "Length:\t\t%d characters\n\n", strlen( string2 ) );

   printf( "Function:\tmemcpy with overlap\n" );
   printf( "Source:\t\t%s\n", string1 + 4 );
   printf( "Destination:\t%s\n", string1 + 10 );
   memcpy( string1 + 10, string1 + 4, 40 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );

}
```

Output

```
Function:       memcpy without overlap
Source:         fox
Destination:    dog jumps over the lazy fox
Result:         The quick brown fox jumps over the lazy fox
Length:         43 characters

Function:       memmove with overlap
Source:         quick brown fox jumps over the lazy dog
Destination:    brown fox jumps over the lazy dog
Result:         The quick quick brown fox jumps over the lazy dog
Length:         49 characters
```

```
Function:       memcpy with overlap
Source:         quick brown dog jumps over the lazy fox
Destination:    brown dog jumps over the lazy fox
Result:         The quick quick quick quick quick quick quick quic
Length:         50 characters
```

Description Set buffers to a specified character.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H (for ANSI compatibility) or MEMORY.H

void *memset(void ***dest*****, int** ***c*****, size_t** ***count*** **);**

void _far * _far _fmemset(void _far ***dest*****, int** ***c*****, size_t** ***count*** **);**

dest	Pointer to destination
c	Character to set
count	Number of characters

Remarks The **memset** and **_fmemset** functions set the first *count* bytes of *dest* to the character *c*.

The **_fmemset** function is a model-independent (large-model) form of the **memset** function. It can be called from any point in any program.

There is a semantic difference between the function version of **memset** and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value The **memset** and **_fmemset** functions return a pointer to *dest*.

Compatibility **memset**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fmemset

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **memccpy, memchr, memcmp, memcpy, strnset**

Example

```
/* MEMSET.C: This program uses memset to set the first four bytes
 * of buffer to "*".
 */
```

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

void main()
{
   char buffer[] = "This is a test of the memset function";

   printf( "Before: %s\n", buffer );
   memset( buffer, '*', 4 );
   printf( "After:  %s\n", buffer );
}
```

Output

```
Before: This is a test of the memset function
After:  **** is a test of the memset function
```

Description

Returns the smaller of two values.

#include <stdlib.h>

type **min(** *type a*, *type b* **);**

type	Any numeric data type
a, *b*	Values of any numeric type to be compared

Remarks

The **min** macro compares two values and returns the value of the smaller one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Return Value

The macro returns the smaller of the two arguments.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

max

Example

```
/* MINMAX.C */
#include <stdlib.h>
#include <stdio.h>

void main()
{
   int a = 10;
   int b = 21;

   printf( "The larger of %d and %d is %d\n",  a, b, max( a, b ) );
   printf( "The smaller of %d and %d is %d\n", a, b, min( a, b ) );
}
```

Output

```
The larger of 10 and 21 is 21
The smaller of 10 and 21 is 10
```

Description Creates a new directory.

#include <direct.h>	Required only for function declarations

int mkdir(char **dirname* **);**

dirname	Path name for new directory

Remarks The **mkdir** function creates a new directory with the specified *dirname*. Only one directory can be created at a time, so only the last component of *dirname* can name a new directory.

The **mkdir** function does not do any translation of path-name delimiters. Both DOS and OS/2 accept either "\" or "/" internally as valid delimiters within path names.

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.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **chdir, rmdir**

Example

```
/* MAKEDIR.C */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
```

```
void main()
{
   int result;

   if( mkdir( "\\testtmp" ) == 0 )
   {
      printf( "Directory '\\testtmp' was successfully created\n" );
      system( "dir \\testtmp" );
      if( rmdir( "\\testtmp" ) == 0 )
         printf( "Directory '\\testtmp' was successfully removed\n"  );
      else
         printf( "Problem removing directory '\\testtmp'\n" );
   }
   else
      printf( "Problem creating directory '\\testtmp'\n" );
}
```

Output

```
Directory '\testtmp' was successfully created

 The volume label in drive C is OS2.
 Directory of C:\TESTTMP

.            <DIR>      6-19-89  11:20a
..           <DIR>      6-19-89  11:20a
    2 File(s)   12730368 bytes free
Directory '\testtmp' was successfully removed
```

Description

Creates a unique file name.

#include <io.h> Required only for function declarations

char *mktemp(char ***_template_ **);

template	File-name pattern

Remarks

The **mktemp** function creates a unique file name by modifying the given *template* argument. The *template* argument has the form:

*base*XXXXXX

where *base* is the part of the new file name that you supply, and the X's 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 copies of 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' through 'z'.

Note that the original template is modified by the first call to **mktemp.** If you then call the **mktemp** function again with the same template (i.e., the original one), you will get an error.

The **mktemp** function generates unique file names but does not create or open files.

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.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also fopen, getpid, open, tempnam, tmpfile

Example

```
/* MKTEMP.C: The program uses mktemp to create five unique file names.
 * It opens each file name to ensure that the next name is unique.
 */

#include <io.h>
#include <string.h>
#include <stdio.h>

char *template = "fnXXXXXX";
char *result;
char names[5][9];

void main()
{
   int i;
   FILE *fp;

   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
      {
         if( (fp = fopen( result, "w" )) != NULL )
             printf( "Unique file name is %s\n", result );
         else
             printf( "Cannot open %s\n", result );
         fclose( fp );
      }
   }
}
```

Output

```
Unique file name is fn000686
Unique file name is fna00686
Unique file name is fnb00686
Unique file name is fnc00686
Unique file name is fnd00686
```

Description

Converts the local time to a calendar value.

#include <time.h>

time_t mktime(struct tm ****timeptr* **);**

timeptr	Pointer to time structure

Remarks

The **mktime** function converts the supplied time structure (possibly incomplete) pointed to by *timeptr* into a fully defined structure with "normalized" values and then converts it to a **time_t** calendar time value. The structure for the **tm** 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.

DOS and OS/2 do not accommodate dates prior to 1980. If *timeptr* references a date before January 1, 1980, **mktime** returns –1.

Note that the **gmtime** and **localtime** functions use a single statically allocated buffer for the conversion. If you supply this buffer to **mktime**, the previous contents will be destroyed.

Return Value

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.**

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

asctime, gmtime, localtime, time

Example

```
/* MKTIME.C: The example takes a number of days as input and returns
 * the time, the current date, and the specified number of days.
 */
```

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

void main()
{
   struct tm when;
   time_t now, result;
   int    days;

   time( &now );
   when = *localtime( &now );
   printf( "Current time is %s\n", asctime( &when ) );
   printf( "How many days to look ahead: " );
   scanf( "%d", &days );

   when.tm_mday = when.tm_mday + days;
   if( (result = mktime( &when )) != (time_t)-1 )
      printf( "In %d days the time will be %s\n",
              days, asctime( &when ) );
   else
      perror( "mktime failed" );
}
```

Output

```
Current time is Mon Jun 19 11:45:20 1989

How many days to look ahead: 23
In 23 days the time will be Wed Jul 12 11:45:20 1989
```

Description

Split a floating-point value into a mantissa and an exponent.

#include <math.h>

double modf(double *x*, **double** **intptr* **);**

long double modfl(long double *x*, **long double** **intptr* **);**

x	Floating-point value
intptr	Pointer to stored integer portion

Remarks

The **modf** functions break down the floating-point value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

The **modfl** function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

The **modf** and **modfl** functions return the signed fractional portion of *x*. There is no error return.

Compatibility

modf

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

modfl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

frexp, ldexp

Example

```
/* MODF.C */
#include <math.h>
#include <stdio.h>
```

```
void main()
{
   double x, y, n;

   x = -14.87654321;       /* Divide x into its fractional */
   y = modf( x, &n );      /* and integer parts            */

   printf( "For %f, the fraction is %f and the integer is %.f\n", x, y, n );
}
```

Output

```
For -14.876543, the fraction is -0.876543 and the integer is -14
```

Description

Moves characters to another segment.

#include <memory.h>	Required only for function declarations
#include <string.h>	Use either STRING.H (for ANSI compatibility) or MEMORY.H

void movedata(unsigned int *srcseg*, **unsigned int** *srcoff*, **unsigned int** *destseg*, **unsigned int** *destoff*, **unsigned int** *count* **);**

srcseg	Segment address of source
srcoff	Segment offset of source
destseg	Segment address of destination
destoff	Segment offset of destination
count	Number of bytes

Remarks

The **movedata** function copies *count* bytes from the source address specified by *srcseg*:*srcoff* to the destination address specified by *destseg*:*destoff*.

The **movedata** function was intended to move far data in small-data-model programs. The newer model-independent **_fmemcpy** and **_fmemmove** functions should be used instead of the **movedata** function. In large-data-model programs, the **memcpy** and **memmove** functions can also be used.

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. These occur when part of the destination is the same memory area as part of the source. The **memmove** function correctly handles overlapping moves.

Return Value

None.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also FP_OFF, FP_SEG, memcpy, memmove, segread

Example

```
/* MOVEDATA.C */
#include <memory.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <malloc.h>

char _far *src = "This is a test.";

void main()
{
   char _far *dest;

   if( (dest = _fmalloc( 80 )) != NULL )
   {
      movedata( FP_SEG( src ),  FP_OFF( src ),
                FP_SEG( dest ), FP_OFF( dest ), _fstrlen( src ) + 1 );
      printf( "The source data at %Fp is '%Fs'\n", src, src );
      printf( "The destination data at %Fp is '%Fs'\n", dest, dest );
      _ffree( dest );
   }
}
```

Output

```
The source data at 2D0A:02B8 is 'This is a test.'
The destination data at 3D0B:0016 is 'This is a test.'
```

Description

Move current graphics positions.

#include <graph.h>

struct xycoord _far _moveto(short *x*, **short** *y* **);**

struct _wxycoord _far _moveto_w(double *wx*, **double** *wy* **);**

x, y	View-coordinate point
wx, wy	Window-coordinate point

Remarks

The **_moveto** functions move the current position to the specified point. The **_moveto** function uses the view-coordinate point (*x*, *y*) as the current position. The **_moveto_w** function uses the window-coordinate point (*wx*, *wy*) as the current position. No drawing takes place.

Return Value

The function returns the coordinates of the previous position. The **_moveto** function returns the coordinates in an **xycoord** structure. The **xycoord** structure, defined in GRAPH.H, contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The **_moveto_w** function returns the coordinates in an **_wxycoord** structure, defined in GRAPH.H. The **_wxycoord** structure contains the following elements:

Element	Description
double wx	*x* window coordinate
double wy	*y* window coordinate

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_lineto** functions

Example

```
/* MOVETO.C: This program draws line segments of different colors. */

#include <graph.h>
#include <stdlib.h>
#include <conio.h>

void main()
{
   short x, y, xinc, yinc, color = 1;
   struct videoconfig v;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &v );
   xinc = v.numxpixels / 50;
   yinc = v.numypixels / 50;

   for( x = 0, y = v.numypixels - 1; x < v.numxpixels; x += xinc, y -= yinc )
   {
      _setcolor( color++ % 16 );
      _moveto( x, 0 );
      _lineto( 0, y );
   }
   getch();

   _setvideomode( _DEFAULTMODE );
}
```

Description Return the size of a memory block allocated in the heap.

#include <malloc.h>	Required only for function declarations

size_t _msize(void ***memblock*** **);**

size_t _bmsize(_segment *seg*, **void _based(void)** ****memblock*** **);**

size_t _fmsize(void _far ****memblock*** **);**

size_t _nmsize(void _near ****memblock*** **);**

memblock	Pointer to memory block
seg	Based-heap segment selector

Remarks The **_msize** family of functions returns the size, in bytes, of the memory block allocated by a call to the appropriate version of the **calloc**, **malloc**, or **realloc** functions.

In large data models (compact-, large-, and huge-model programs), **_msize** maps to **_fmsize.** In small data models (tiny-, small-, and medium-model programs), **_msize** maps to **_nmsize.**

The **_nmsize** function returns the size (in bytes) of the memory block allocated by a call to **_nmalloc**, and the **_fmsize** function returns the size (in bytes) of the memory block allocated by a call to **_fmalloc** or **_frealloc.** The **_bmsize** function returns the size of a block allocated in segment *seg* by a call to **_bmalloc**, **_bcalloc**, or **_brealloc.**

The location of the memory block is indicated below:

Function	Data Segment
_msize	Depends on data model of program
_bmsize	Based heap segment specified by *seg* value
_fmsize	Far heap segment (outside default data segment)
_nmsize	Default data segment (inside near heap)

Return Value All four functions return the size (in bytes) as an unsigned integer.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **calloc** functions, **_expand** functions, **malloc** functions, **realloc** functions

Example

```
/* REALLOC.C: This program allocates a block of memory for buffer
 * and then uses _msize to display the size of that block. Next, it
 * uses realloc to expand the amount of memory used by buffer
 * and then calls _msize again to display the new amount of
 * memory allocated to buffer.
 */

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

void main()
{
   long *buffer;
   size_t size;

   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );

   size = _msize( buffer );
   printf( "Size of block after malloc of 1000 longs: %u\n", size );

   /* Reallocate and show new size: */
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) )) == NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after realloc of 1000 more longs: %u\n", size );

   free( buffer );
}
```

Output

```
Size of block after malloc of 1000 longs: 4000
Size of block after realloc of 1000 more longs: 8000
```

Description

Registers a routine to be called at exit time.

#include <stdlib.h>

onexit_t onexit(onexit_t *func* **);**

func	Pointer to function to be called at exit

Remarks

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 is executed in LIFO (last-in–first-out) order. 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.

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 as **onexit**, and should be used instead of **onexit** when ANSI portability is desired.

All routines passed to **onexit** should have the **_loadds** attribute if used in multithread dynamic-link libraries.

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.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

exit

Example

```
/* ONEXIT.C */
#include <stdlib.h>
#include <stdio.h>

/* Prototypes */
void fn1( void ), fn2( void ), fn3( void ),  fn4( void );
```

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

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

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

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

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

Output

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

Description

Opens a file.

#include <fcntl.h>

#include <sys\types.h>

#include <sys\stat.h>

#include <io.h> Required only for function declarations

int open(char **filename*, **int** *oflag* **[[, int** *pmode***]]);**

filename	File name
oflag	Type of operations allowed
pmode	Permission mode

Remarks

The **open** function opens the file specified by *filename* and prepares the file for subsequent reading or writing, as defined by *oflag*. The *oflag* argument is an integer expression formed from one or more of the manifest constants defined in FCNTL.H (listed below). When two or more manifest constants are used to form the *oflag* argument, the constants are combined with the bitwise-OR operator (|). See Section 2.5, "File Handling," for a discussion of binary and text modes.

The FCNTL.H file defines the following manifest constants:

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.
O_CREAT	Creates and opens a new file for writing; this has no effect if the file specified by *filename* exists.
O_EXCL	Returns an error value if the file specified by *filename* 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.

O_TRUNC	Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed. If this flag is given, you cannot specify **O_RDONLY**.
O_WRONLY	Opens file for writing only; if this flag is given, neither **O_RDONLY** nor **O_RDWR** can be given.

WARNING *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. There is no default value for the access mode.

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 DOS and OS/2, 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.

The **open** function applies the current file-permission mask to *pmode* before setting the permissions (see **umask**).

The *filename* argument used in the **open** function is affected by the DOS APPEND command.

Note that under DOS versions 3.0 and later, a problem occurs when **SHARE** is installed and a new file is opened with *oflag* set to **O_CREAT | O_RDONLY** or **O_CREAT | O _WRONLY** and *pmode* set to **S_IREAD.** Under these conditions, the operating system prematurely closes the file during system calls made within **open.** This problem does not occur under OS/2.

To work around the problem, open the file with the *pmode* argument set to **S_IWRITE.** Then close the file and use **chmod** to change the access mode back to **S_IREAD.** Another work-around is to open the file with *pmode* set to **S_IREAD** and *oflag* set to **O_CREAT | O_RDWR.**

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.
EINVAL	An invalid *oflag* or *pmode* argument was given.
EMFILE	No more file handles available (too many open files).
ENOENT	File or path name not found.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

access, chmod, close, creat, dup, dup2, fopen, sopen, umask

Example

```
/* OPEN.C: This program uses open to open a file named OPEN.C for input
 * and a file named OPEN.OUT for output. The files are then closed.
 */

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

void main()
{
   int fh1, fh2;
```

```
    fh1 = open( "OPEN.C", O_RDONLY );
    if( fh1 == -1 )
       perror( "open failed on input file" );
    else
    {
       printf( "open succeeded on input file\n" );
       close( fh1 );
    }

    fh2 = open( "OPEN.OUT", 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" );
       close( fh2 );
    }
}
```

Output

```
open succeeded on input file
open succeeded on output file
```

Description Prints font-based text in graphics mode.

#include <graph.h>

void _far_outgtext(unsigned char _far ****text* **);**

text	Text string to output

Remarks The **_outgtext** function outputs on the screen the null-terminated string that *text* points to. The text is output using the current font at the current graphics position and in the current color.

No formatting is provided, in contrast to the standard console I/O library routines such as **printf.**

After it outputs the text, **_outgtext** updates the current graphics position.

The **_outgtext** function operates only in graphics video modes (e.g., **_MRES4COLOR**). Because it is a graphics function, the color of text is set by the **_setcolor** function, not by the **_settextcolor** function.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_moveto** functions, **_setcolor**, **_setfont**

Example

```
/* OUTGTXT.C illustrates font output using functions:
 *    _registerfonts        _setfont           _outgtext
 *    _unregisterfonts      _getfontinfo       _getgtextextent
 *    _setgtextvector
 */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <graph.h>
```

```
#define NFONTS 6

unsigned char *face[NFONTS] =
{
    "Courier", "Helvetica", "Times Roman", "Modern", "Script", "Roman"
};
unsigned char *options[NFONTS] =
{
    "courier", "helv", "tms rmn", "modern", "script", "roman"
};

void main()
{
    unsigned char list[20];
    char fondir[_MAX_PATH];
    struct videoconfig vc;
    struct _fontinfo fi;
    short fontnum, x, y;

    /* Read header info from all .FON files in current or given directory. */
    if( _registerfonts( "*.FON" ) <= 0 )
    {
        _outtext( "Enter full path where .FON files are located: " );
        gets( fondir );
        strcat( fondir, "\\*.FON" );
        if( _registerfonts( fondir ) <= 0 )
        {
            _outtext( "Error: can't register fonts" );
            exit( 1 );
        }
    }

    /* Set highest available graphics mode and get configuration. */
    if( !_setvideomode( _MAXRESMODE ) )
        exit( 1 );
    _getvideoconfig( &vc );

    /* Display each font name centered on screen. */
    for( fontnum = 0; fontnum < NFONTS; fontnum++ )
    {
        /* Build options string. */
        strcat( strcat( strcpy( list, "t'" ), options[fontnum] ), "'");
        strcat( list, "h30w24b" );

        _clearscreen( _GCLEARSCREEN );
        if( _setfont( list ) >= 0 )
        {
```

```
            /* Use length of text and height of font to center text. */
            x = (vc.numxpixels / 2) - (_getgtextextent( face[fontnum] ) / 2);
            y = (vc.numypixels / 2) + (_getgtextextent( face[fontnum] ) / 2);
            if( _getfontinfo( &fi ) )
            {
                _outtext( "Error: Can't get font information" );
                break;
            }
            _moveto( x, y );
            if( vc.numcolors > 2 )
                _setcolor( fontnum + 2 );

            /* Rotate and display text. */
            _setgtextvector( 1, 0 );
            _outgtext( face[fontnum] );
            _setgtextvector( 0, 1 );
            _outgtext( face[fontnum] );
            _setgtextvector( -1, 0 );
            _outgtext( face[fontnum] );
            _setgtextvector( 0, -1 );
            _outgtext( face[fontnum] );
        }
        else
        {
            _outtext( "Error: Can't set font: " );
            _outtext( list );
        }
        getch();
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

Description Prints text of a specified length in graphics mode.

#include <graph.h>

void _far _outmem(unsigned char _far ***text*****, short** ***length*** **);**

text	Text string to output
length	Length of string to output

Remarks The **_outmem** function outputs the string that *text* points to. The *length* argument specifies the number of characters to output.

Unlike **_outtext**, the **_outmem** function prints all characters literally, including ASCII 10, 13, and 0 as the equivalent graphics characters. No formatting is provided. Text is printed using the current text color, starting at the current text position.

To output text using special fonts, you must use the **_outgtext** function.

Return Value None.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_outtext, _settextcolor, _settextposition, _settextwindow**

Example

```
/* OUTMEM.C illustrates:
 *    _outmem
 */

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

void main()
{
    int  i, len;
    char tmp[10];
```

```
    _clearscreen( _GCLEARSCREEN );
    for( i = 0; i < 256; i++ )
    {
        _settextposition( (i % 24) + 1, (i / 24) * 7 );
        len = sprintf( tmp, "%3d %c", i, i );
        _outmem( tmp, len );
    }
    _settextposition( 24, 1 );
}
```

Description

Outputs a byte (**outp**) or a word (**outpw**) at a port.

#include <conio.h>	Required only for function declarations

int outp(unsigned *port*, **int** *databyte* **);**

unsigned outpw(unsigned *port*, **unsigned** *dataword* **);**

port	Port number
databyte	Output value
dataword	Output value

Remarks

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 *dataword* can be any value in the range 0 – 65,535.

Both **outp** and **outpw** are supported in OS/2. You must use a .DEF file to declare the IOSEG segment the run-time library uses to perform input/output on the port. In addition, the intrinsic (/Oi) versions of these functions do not work unless you put the code in a segment that is marked with the **IOPL** keyword in the .DEF file.

You cannot do IOPL from a regular code segment, so the run-time library has declared a separate code segment called **_IOSEG**. In order to use **inp**, **inpw**, **outp**, or **outp** in any of the protected mode run-time libraries (?LIBCP, LLIBCDLL, LLIBCMT, or CDLLOBJS-based DLL), you must have a .DEF file with this line in it:

```
SEGMENTS _IOSEG CLASS 'IOSEG_CODE' IOPL
```

Return Value

The functions return the data output. There is no error return.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

inp, inpw

Example

```
/* OUTP.C: This program uses inp and outp to make sound of variable tone
 * and duration.
 */
```

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

void Beep( unsigned duration, unsigned frequency ); /* Prototypes */
void Sleep( clock_t wait );

void main ()
{
    Beep( 698, 700 );
    Beep( 523, 500 );
}

/* Sounds the speaker for a time specified in microseconds by duration
 * at a pitch specified in hertz by frequency.
 */
void Beep( unsigned frequency, unsigned duration )
{
    int control;

    /* If frequency is 0, Beep doesn't try to make a sound. */
    if( frequency )
    {
        /* 75 is about the shortest reliable duration of a sound. */
        if( duration < 75 )
            duration = 75;

        /* Prepare timer by sending 10111100 to port 43. */
        outp( 0x43, 0xb6 );

        /* Divide input frequency by timer ticks per second and
         * write (byte by byte) to timer.
         */
        frequency = (unsigned)(1193180L / frequency);
        outp( 0x42, (char)frequency );
        outp( 0x42, (char)(frequency >> 8) );

        /* Save speaker control byte. */
        control = inp( 0x61 );

        /* Turn on the speaker (with bits 0 and 1). */
        outp( 0x61, control | 0x3 );
    }

    Sleep( (clock_t)duration );

    /* Turn speaker back on if necessary. */
    if( frequency )
        outp( 0x61, control );
}
```

```
/* Pauses for a specified number of microseconds. */
void Sleep( clock_t wait )
{
    clock_t goal;

    goal = wait + clock();
    while( goal > clock() )
        ;
}
```

Description Prints text in graphics mode.

#include <graph.h>

void _far _outtext(unsigned char _far *_text_);

text	Text string to output

Remarks 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**. This function will work in any screen mode.

Text output begins at the current text position.

To output text using special fonts, you must use the **_outgtext** function.

Return Value None.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_outmem, _settextcolor, _settextposition, _settextwindow**

Example

```
/* OUTTXT.C: This example illustrates text output functions:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition
 */

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

char buffer [80];

void main()
{

   /* Save original foreground, background, and text position */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct rccoord oldpos;
```

```
   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );

   /* First time no blink, second time blinking. */
   for( blink = 0; blink <= 16; blink += 16 )
   {
      /* Loop through 8 background colors. */
      for( bgd = 0; bgd < 8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf(buffer, "Back: %d Fore:", bgd );
         _outtext( buffer );

         /* Loop through 16 foreground colors. */
         for( fgd = 0; fgd < 16; fgd++ )
         {
            _settextcolor( fgd + blink );
            sprintf( buffer, " %2d ", fgd + blink );
            _outtext( buffer );
         }
      }
   }
   getch();

   /* Restore original foreground, background, and text position. */
   _settextcolor( oldfgd );
   _setbkcolor( oldbgd );
   _clearscreen( _GCLEARSCREEN );
   _settextposition( oldpos.row, oldpos.col );
}
```

Description Waits for a child command and closes the stream on the associated pipe.

#include <stdio.h> Function declaration

int _pclose(FILE ***stream*** **);**

stream File stream returned by previous call to **_popen**

Remarks The **_pclose** function waits for a child command and closes the stream on the associated pipe. The argument *stream* is the return value from a previous call to **_popen**. The **_pclose** function looks up the process ID of the child command started by the associated **_popen** call, closes the stream, executes a **cwait** call on the child command, and returns the exit status of the child command. See **_pipe** for a general discussion of pipes in OS/2.

Return Value The **_pclose** function returns the exit status of the child command. The format of the return value is the same as that for **cwait**, with the exception that the low-order and high-order bytes are swapped. If an error occurs, –1 is returned.

Compatibility □ ANSI □ DOS ■ OS/2 ■ UNIX ■ XENIX

A similar function (**pclose**) is available in the XENIX and UNIX operating environments.

See Also **cwait, _pipe, _popen**

Example See the example for **_popen**.

Description Prints an error message.

#include <stdio.h> Required only for function declarations

void perror(const char ***string*** **);**

string String message to print

Remarks The **perror** function prints an error message to **stderr**. The *string* argument is printed first, followed by a colon, then by the system error message for the last library call that produced the error, and finally by a newline 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** (defined in ERRNO.H). 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.

Under DOS and OS/2, some of the **errno** values listed in ERRNO.H are not used. These additional **errno** values are reserved for UNIX and XENIX use. See Section 3.3, "**_doserrno, errno, sys_errlist, sys_nerr,**" for a list of **errno** values used on DOS and OS/2 and the corresponding error messages. The **perror** function prints an empty string for any **errno** value not used under the operating system.

Return Value None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **clearerr, ferror, strerror**

Example

```
/* PERROR.C: This program attempts to open a file named NOSUCHF.ILE.
 * Since this file probably doesn't exist, an error message is displayed.
 * The same message is created using perror, strerror, and _strerror.
 */
```

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

void main()
{
   int  fh;

   if( (fh = open( "NOSUCHF.ILE", O_RDONLY )) == -1 )
   {
      /* Three ways to create error message: */
      perror( "perror says open failed" );
      printf( "strerror says open failed: %s\n", strerror( errno ) );
      printf( _strerror( "_strerror says open failed" ) );
   }
   else
   {
      printf( "open succeeded on input file\n" );
      close( fh );
   }
}
```

Output

```
perror says open failed: No such file or directory
strerror says open failed: No such file or directory
_strerror says open failed: No such file or directory
```

Description Analyze a series of data.

#include <pgchart.h>

short _far _pg_analyzechart(chartenv _far ****env*, **char _far * _far** ****categories*,
float _far ****values*, **short** *n* **);**

short _far _pg_analyzechartms(chartenv _far ****env*, **char _far * _far** ****categories*,
float _far ****values*, **short** *nseries*, **short** *n*, **short** *arraydim*,
char _far * _far ****serieslabels* **);**

env	Chart environment variable
categories	Array of category variables
values	Array of data values
nseries	Number of series to chart
n	Number of data values to chart
arraydim	Row dimension of data array
serieslabels	Array of labels for series

Remarks The **_pg_analyzechart** routines analyze a single or multiple series of data without actually displaying the presentation-graphic image.

The **_pg_analyzechart** function fills the chart environment with default values for a single-series bar, column, or line chart, depending on the type specified by the call to the **_pg_defaultchart** function. The variables calculated by **_pg_analyzechart** reflect the data given in the arguments *categories* and *values*. All arguments are the same as those used with the **_pg_chart** function.

The **_pg_analyzechartms** function fills the chart environment with default values for a multiseries bar, column, or line chart, depending on which type is specified in the **_pg_defaultchart** function. The variables calculated by **_pg_analyzechartms** reflect the data given in the arguments *categories* and *values*. All arguments are the same as those used with the **_pg_chartms** function.

Boolean flags in the chart environment, such as **AUTOSCALE** and **LEGEND**, should be set to **TRUE** before calling either **_pg_analyzechart** function. This will ensure that the function will calculate all defaults.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_analyzechart** and **_pg_analyzechartms** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_chart** functions, **_pg_defaultchart**, **_pg_initchart**

Example

```
/* PGACHART.C: This example illustrates presentation-graphics
 * analyze functions.
 * The example uses
 *      _pg_analyzechartms
 * The same principles apply for
 *      _pg_analyzepie          _pg_analyzechart
 *      _pg_analyzescatter      _pg_analyzescatterms
 */

#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>

#define FALSE  0
#define TRUE   1

/* Note data declared as a single-dimension array. The multiseries
 * chart functions expect only one dimension. See _pg_chartms
 * example for alternate method using multidimension array.
 */
#define TEAMS  4
#define MONTHS 3
float _far values[TEAMS * MONTHS] = { .435,   .522,   .671,
                                      .533,   .431,   .590,
                                      .723,   .624,   .488,
                                      .329,   .226,   .401   };
char _far *months[MONTHS] =         { "May",  "June", "July" };
char _far *teams[TEAMS] = { "Reds", "Sox", "Cubs", "Mets" };

void main()
{
   chartenv env;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _pg_initchart();                    /* Initialize chart system.   */
```

```
   /* Default multiseries bar chart */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Little League Records - Default" );
   _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
   getch();
    _clearscreen( _GCLEARSCREEN );

   /* Analyze multiseries bar chart with autoscale. This sets all
    * default scale values. We want y axis values to be automatic.
    */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Little League Records - Customized" );
   env.xaxis.autoscale = TRUE;
   _pg_analyzechartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );

   /* Now customize some of the x axis values. Then draw the chart.  */
   env.xaxis.autoscale = FALSE;
   env.xaxis.scalemax = 1.0;           /* Make scale show 0.0 to 1.0.   */
   env.xaxis.ticinterval = 0.2;        /* Don't make scale too crowded. */
   env.xaxis.ticdecimals = 3;          /* Show three decimals.          */
   strcpy( env.xaxis.scaletitle.title, "Win/Loss Percentage" );
   _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
   getch();

   _setvideomode( _DEFAULTMODE );
}
```

Description Analyzes a single series of data for a pie chart.

#include <pgchart.h>

short _far _pg_analyzepie(chartenv _far **env*, **char _far * _far** **categories*,
float _far **values*, **short _far** **explode*, **short** *n* **);**

env	Chart environment variable
categories	Array of category variables
values	Array of data values
explode	Array of explode flags
n	Number of data values to chart

Remarks The **_pg_analyzepie** function analyzes a single series of data without actually displaying the graphic image.

The **_pg_analyzepie** function fills the chart environment for a pie chart using the data contained in the array *values*. All arguments are the same as those used in the **_pg_chartpie** function.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_analyzepie** function returns 0 if there were no errors. A nonzero value indicates a failure.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_chartpie, _pg_defaultchart, _pg_initchart**

Example See the example for **_pg_analyzechart.**

Description Analyze a series of data for a scatter chart.

#include <pgchart.h>

short _far _pg_analyzescatter(chartenv _far **env*, **float _far** **xvalues*,
  float _far **yvalues*, **short** *n* **);**

short _far _pg_analyzescatterms(chartenv _far **env*, **float _far** **xvalues*,
  float _far **yvalues*, **short** *nseries*, **short** *n*, **short** *rowdim*,
  char _far * **_far** **serieslabels* **);**

env	Chart environment structure
xvalues	Array of *x*-axis data values
yvalues	Array of *y*-axis data values
n	Number of data values to chart
nseries	Number of series to chart
rowdim	Row dimension of data array
serieslabels	Array of labels for series

Remarks The **_pg_analyzescatter** set of routines analyzes a single or multiple series of data without actually displaying the graphic image.

The **_pg_analyzescatter** function fills the chart environment for a single-series scatter diagram. The variables calculated by this function reflect the data given in the arguments *xvalues* and *yvalues*. All arguments are the same as those used in the **_pg_chartscatter** function.

The **_pg_analyzescatterms** function fills the chart environment for a multiseries scatter diagram. The variables calculated by **_pg_analyzescatterms** reflect the data given in the arguments *xvalues* and *yvalues*. All arguments are the same as those used in the function **_pg_chartscatterms**.

Boolean flags in the chart environment, such as **AUTOSCALE** and **LEGEND**, should be set to **TRUE** before calling **_pg_analyzescatterms**; this ensures that the function will calculate all defaults.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_analyzescatter** and **_pg_analyzescatterms** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_chartscatter** functions, **_pg_defaultchart**, **_pg_initchart**

Example See the example for **_pg_analyzechart**.

Description

Display single-series or multiseries charts.

#include <pgchart.h>

short _far _pg_chart(chartenv _far ***env*****, char _far * _far** ***categories*****,**
float _far ***values*****, short** ***n*** **);**

short _far _pg_chartms(chartenv _far ***env*****, char _far * _far** ***categories*****,**
float _far ***values*****, short** ***nseries*****, short** ***n*****, short** ***arraydim*****,**
char _far * _far ***serieslabels*** **);**

env	Chart environment variable
categories	Array of category variables
values	Array of data values
n	Number of data values to chart
nseries	Number of series to chart
arraydim	Row dimension of data array
serieslabels	Array of labels for series

Remarks

The **_pg_chart** function displays a single-series bar, column, or line chart, depending on the type specified in the chart environment variable (*env*).

The **_pg_chartms** function displays a multiseries bar, column, or line chart, depending on the type specified in the chart environment. All the series must contain the same number of data points, specified by the argument *n*.

The array *values* is a two-dimensional array containing all value data for every series to be plotted on the chart. Each column of *values* represents a single series. The parameter *rowdim* is the integer value used to dimension rows in the array declaration for *values*.

For example, the following code fragment declares the identifier `values` to be a two-dimensional floating-point array with 20 rows and 10 columns:

```
#define ARRAYDIM 20
float values [ARRAYDIM][10];
short rowdim = ARRAYDIM;
```

Note that the number of columns in the *values* array cannot exceed 10, the maximum number of data series on a single chart. Note also that `rowdim` must be greater than or equal to the argument *n*, and the column dimension in the array declaration must be greater than or equal to the argument *nseries*. If *n* and *nseries* are set to values less than the full dimensional size of the *values* array, only part of the data contained in *values* will be plotted.

The array *serieslabels* holds the labels used in the chart legend to identify each series.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_chart** and **_pg_chartms** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_analyzechart** functions, **_pg_defaultchart**, **_pg_initchart**

Example

```
/* PGCHART.C: This example illustrates presentation-graphics support
 * routines and single-series chart routines, including
 *    _pg_initchart   _pg_defaultchart   _pg_chart   _pg_chartpie
 */

#include <conio.h>
#include <graph.h>
#include <string.h>
#include <stdlib.h>
#include <pgchart.h>

#define COUNTRIES 5
float _far value[COUNTRIES] =    { 42.5,    14.3,    35.2,    21.3,    32.6    };
char _far *category[COUNTRIES] = { "USSR",  "France","USA",   "UK",    "Other" };
short _far explode[COUNTRIES] =  { 0,       1,       0,       1,       0       };

void main()
{
   chartenv env;
   short mode = _VRES16COLOR;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _pg_initchart();                     /* Initialize chart system. */

   /* Single-series bar chart */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Widget Production" );
   _pg_chart( &env, category, value, COUNTRIES );
   getch();
   _clearscreen( _GCLEARSCREEN );
```

```
    /* Single-series column chart */
    _pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Widget Production" );
    _pg_chart( &env, category, value, COUNTRIES );
    getch();
    _clearscreen( _GCLEARSCREEN );

    /* Pie chart */
    _pg_defaultchart( &env, _PG_PIECHART, _PG_PERCENT );
    strcpy( env.maintitle.title, "Widget Production" );
    _pg_chartpie( &env, category, value, explode, COUNTRIES );
    getch();

    _setvideomode( _DEFAULTMODE );
}
```

Description Displays a pie chart.

#include <pgchart.h>

short _far _pg_chartpie(chartenv _far ***env*****, char _far * _far** ***categories*****,**
float _far ***values*****, short _far** ***explode*****, short** ***n*** **);**

env	Chart environment structure
categories	Array of category labels
values	Array of data values
explode	Array of explode flags
n	Number of data values to chart

Remarks The **_pg_chartpie** function displays a pie chart for the data contained in the array *values*. Pie charts are formed from a single series of data—there is no multiseries version of pie charts as there is for other chart types.

The array *explode* must be dimensioned so that its length is greater than or equal to the argument *n*. All entries in *explode* are either 0 or 1. If an entry is 1, the corresponding pie slice is displayed slightly removed from the rest of the pie.

For example, if the *explode* array is initialized as

```
short explode[5] = {0, 1, 0, 0, 0};
```

the pie slice corresponding to the second entry of the *categories* array will be displayed "exploded" from the other four slices.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_chartpie** function returns 0 if there were no errors. A nonzero value indicates a failure.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_analyzepie, _pg_defaultchart, _pg_initchart**

Example See the example for **_pg_chart.**

Description Display scatter charts.

#include <pgchart.h>

short _far _pg_chartscatter(chartenv _far ****env*, float _far ****xvalues***,**
float _far ****yvalues*, short** ***n*** **);**

short _far _pg_chartscatterms(chartenv _far ****env*, float _far ****xvalues***,**
float _far ****yvalues*, short** ***nseries***, **short** ***n***, **short** ***rowdim***,
char _far * _far ****serieslabels* **);

env	Chart environment structure
xvalues	Array of *x*-axis data values
yvalues	Array of *y*-axis data values
n	Number of data values to chart
nseries	Number of series to chart
rowdim	Row dimension of data array
serieslabels	Array of labels for series

Remarks The **_pg_chartscatter** function displays a scatter diagram for a single series of data.

The **_pg_chartscatterms** function displays a scatter diagram for more than one series of data.

The arguments *xvalues* and *yvalues* are two-dimensional arrays containing data for the *x* axis and *y* axis, respectively. Columns for each array hold data for individual series; thus the first columns of *xvalues* and *yvalues* contain plot data for the first series, the second columns contain plot data for the second series, and so forth.

The *n*, *rowdim*, *nseries*, and *serieslabels* arguments fulfill the same purposes as those used in the **_pg_chartms** function. See **_pg_chartms** for an explanation of these arguments.

For a discussion of the chart environment and related topics, see Section 2.6.2, "Presentation-Graphics Functions."

Return Value The **_pg_chartscatter** and **_pg_chartscatterms** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_analyzescatter** functions, **_pg_defaultchart**, **_pg_initchart**

Example See the example for **_pg_chart**.

Description

Initializes the chart environment.

#include <pgchart.h>

short _far _pg_defaultchart(chartenv _far **env*, **short** *charttype*, **short** *chartstyle* **);**

env	Chart environment structure
charttype	Chart type
chartstyle	Chart style

Remarks

The **_pg_defaultchart** function initializes all necessary variables in the chart environment for the chart type by the variable *charttype*.

All title fields in the environment structure are blanked. Titles should be set in the proper fields after calling **_pg_defaultchart**.

The *charttype* variable can be set to one of the following manifest constants:

Chart Type	Description
_PG_BARCHART	Bar chart
_PG_COLUMNCHART	Column chart
_PG_LINECHART	Line chart
_PG_PIECHART	Pie chart
_PG_SCATTERCHART	Scatter chart

The *chartstyle* variable specifies the style of the chart with either the number "1" or the number "2." Each of the five types of presentation-graphics charts can appear in two different chart styles, as described below:

Chart Type	Chart Style 1	Chart Style 2
Bar	Side by side	Stacked
Column	Side by side	Stacked
Line	Points with lines	Points only
Pie	Percent	No percent
Scatter	Points with lines	Points only

In a pie chart, the pieces are "exploded" according to the *explode* array argument in the **_pg_chartpie** function. In the "percent" format, percentages are printed next to each slice. Bar and column charts have only one style when displaying a single series of data. The styles "side by side" and "stacked" are applicable only when more than one series appear on the same chart. The first style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The stacked style emphasizes relative sizes between bars and columns.

Return Value

The **_pg_defaultchart** function returns 0 if there were no errors. A nonzero value indicates a failure.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_pg_getchardef, _pg_getpalette, _pg_getstyleset, _pg_hlabelchart, _pg_initchart, _pg_resetpalette, _pg_resetstyleset, _pg_setchardef, _pg_setpalette, _pg_setstyleset, _pg_vlabelchart

Example

See the example for **_pg_chart**.

Description Gets the pixel bit map for the specified character.

#include <pgchart.h>

short _far _pg_getchardef(short *charnum*, **unsigned char _far** *****chardef* **);**

charnum	ASCII number of character
chardef	Pointer to 8-by-8 bit map array

Remarks The **_pg_getchardef** function retrieves the current 8-by-8 pixel bit map for the character having the ASCII number *charnum*. The bit map is stored in the *chardef* array.

Return Value The **_pg_getchardef** function returns 0 if there were no errors. A nonzero value indicates an error.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_defaultchart, _pg_initchart, _pg_setchardef**

Description

Gets palette colors, line styles, and patterns.

#include <pgchart.h>

short _far _pg_getpalette(paletteentry _far ****palette* **);**

palette	Pointer to first palette structure in array

Remarks

The **_pg_getpalette** function retrieves palette colors, line styles, fill patterns, and plot characters for all palettes. The pointer *palette* points to an array of palette structures that will contain the desired palette values.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value

The function **_pg_getpalette** returns 0 if there were no errors, and it returns the value **_BADSCREENMODE** if current palettes have not been initialized by a previous call to **_pg_setpalette.**

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_pg_defaultchart, _pg_initchart, _pg_resetpalette, _pg_setpalette

Example

```
/* PGGPAL.C: This example illustrates presentation-graphics palettes
 * and the routines that modify them, including
 *    _pg_getpalette    _pg_resetpalette    _pg_setstyleset
 *    _pg_getstyleset   _pg_resetstyleset   _pg_vlabelchart
 *    _pg_hlabelchart   _pg_setpalette
 */
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>
```

```
#define TEAMS  2
#define MONTHS 3
float _far values[TEAMS][MONTHS] = { { .435,   .522,   .671  },
                                     { .533,   .431,   .401  } };
char _far *months[MONTHS] =          { "May", "June", "July" };
char _far *teams[TEAMS] = { "Cubs", "Reds" };

fillmap fill1 = { 0x99, 0x33, 0x66, 0xcc, 0x99, 0x33, 0x66, 0xcc };
fillmap fill2 = { 0x99, 0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33 };
styleset styles;
palettetype pal;

void main()
{
   chartenv env;
   short mode = _VRES16COLOR;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _pg_initchart();                      /* Initialize chart system.   */

   /* Modify global set of line styles used for borders, grids, and
    * data connectors. Note that this change is used before
    * _pg_defaultchart, which will use the style set.
    */
   _pg_getstyleset( styles );            /* Get styles and modify      */
   styles[1] = 0x5555;                   /*   style 1 (used for        */
   _pg_setstyleset( styles );            /*   borders)-then set new.   */

   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );

   /* Modify palette for data lines, colors, fill patterns, and
    * characters. Note that the line styles are set in the palette, not
    * in the style set, so that only data connectors will be affected.
    */
   _pg_getpalette( pal );                /* Get default palette.        */
   pal[1].plotchar = 16;                 /* Set to ASCII 16 and 17.     */
   pal[2].plotchar = 17;
   memcpy( pal[1].fill, fill1, 8 );      /* Copy fill masks to palette. */
   memcpy( pal[2].fill, fill2, 8 );
   pal[1].color = 3;                     /* Change palette colors.      */
   pal[2].color = 4;
   pal[1].style = 0xfcfc;                /* Change palette line styles. */
   pal[2].style = 0x0303;
   _pg_setpalette( pal );                /* Put modified palette.       */
```

```
    /* Multiseries bar chart */
    strcpy( env.maintitle.title, "Little League Records - Customized" );
    _pg_chartms( &env, months, (float _far *)values,
              TEAMS, MONTHS, MONTHS, teams );
    getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart */
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Little League Records - Customized" );
    _pg_chartms( &env, months, (float _far *)values,
                  TEAMS, MONTHS, MONTHS, teams );

    /* Print labels. */
    _pg_hlabelchart( &env, (short)(env.chartwindow.x2 * .75),
                           (short)(env.chartwindow.y2 * .10),
                           12, "Up and up!" );
    _pg_vlabelchart( &env, (short)(env.chartwindow.x2 * .75),
                           (short)(env.chartwindow.y2 * .45),
                           13, "Sliding down!" );
    getch();
    _clearscreen( _GCLEARSCREEN );

    _pg_resetpalette();                    /* Restore default palette    */
    _pg_resetstyleset();                   /*   and style set.           */

    /* Multiseries bar chart */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records - Default" );
    _pg_chartms( &env, months, (float _far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart */
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Little League Records - Default" );
    _pg_chartms( &env, months, (float _far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    getch();

    _setvideomode( _DEFAULTMODE );
}
```

Description Gets the current styleset.

#include <pgchart.h>

void _far _pg_getstyleset(unsigned short _far ****styleset* **);

styleset	Pointer to current styleset

Remarks The **_pg_getstyleset** function retrieves the contents of the current styleset.

Return Value None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_defaultchart, _pg_initchart, _pg_resetstyleset, _pg_setstyleset**

Example See the example for **_pg_getpalette**.

Description Writes text horizontally on the screen.

#include <pgchart.h>

short _far _pg_hlabelchart(chartenv _far **env*, **short** *x*, **short** *y*, **short** *color*,
char _far **label* **);**

env	Chart environment structure
x	*x*-coordinate for text
y	Pixel *y*-coordinate for text
color	Color code for text
label	Label text

Remarks The **_pg_hlabelchart** function writes text horizontally on the screen. The arguments *x* and *y* are pixel coordinates for the beginning location of text relative to the upper-left corner of the chart window.

Return Value The **_pg_hlabelchart** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_defaultchart, _pg_initchart, _pg_vlabelchart**

Example See the example for **_pg_getpalette.**

Description

Initializes presentation graphics.

#include <pgchart.h>

short _far _pg_initchart(void);

Remarks

The **_pg_initchart** function initializes the presentation-graphics package. It initializes the color and style pools, resets the chartline styleset, builds default palette modes, and reads the presentation-graphics font definition from the disk. This function is required in all programs that use presentation graphics. The **_pg_initchart** function must be called before any of the other functions in the presentation-graphics library.

The **_pg_initchart** function assumes a valid graphics mode has been established. Therefore, it must be called only after a successful call to the library function **_setvideomode**.

Return Value

The **_pg_initchart** functions return 0 if there were no errors. A nonzero value indicates a failure.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_pg_defaultchart, _pg_getchardef, _pg_getpalette, _pg_getstyleset, _pg_hlabelchart, _pg_resetpalette, _resetstyleset, _pg_setchardef, _pg_setpalette, _pg_setstyleset, _pg_vlabelchart, _setvideomode

Example

See the example for **_pg_chart**.

Description

Resets palette colors, line styles, and patterns to default values.

#include <pgchart.h>

short _far _pg_resetpalette(void);

Remarks

The **_pg_resetpalette** function sets the palette colors, line styles, fill patterns, and plot characters for the palette to the default for the current screen mode.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value

The **_pg_resetpalette** function returns 0 if there were no errors. If the screen mode is not valid, the value **_BADSCREENMODE** is returned.

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

_pg_defaultchart, _pg_getpalette, _pg_initchart, _pg_setpalette

Example

See the example for **_pg_getpalette.**

Description Resets styleset to default values.

#include <pgchart.h>

void _far _pg_resetstyleset(void);

Remarks The **_pg_resetstyleset** function reinitializes the styleset to the default values for the current screen mode.

Return Value None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_defaultchart, _pg_getstyleset, _pg_initchart, _pg_setstyleset**

Example See the example for **_pg_getpalette**.

Description

Sets the pixel bit map for the specified character.

#include <pgchart.h>

short _far _pg_setchardef(short *charnum*, **unsigned char _far** **chardef* **);**

charnum	ASCII number of character
chardef	Pointer to an 8-by-8 bit map array for the character

Remarks

The **_pg_setchardef** function sets the 8-by-8 pixel bit map for the character with the ASCII number *charnum*. The bit map is stored in the *chardef* array.

Return Value

The **_pg_setchardef** function returns 0 if there was no error. A nonzero value indicates an error.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_pg_defaultchart, _pg_getchardef, _pg_initchart

Description Sets palette colors, line styles, and patterns.

#include <pgchart.h>

short _far _pg_setpalette(paletteentry _far ***palette*);**

palette	Pointer to first palette structure in array

Remarks The **_pg_setpalette** function sets palette colors, line styles, fill patterns, and plot characters for all palettes. The pointer *palette* points to an array of palette structures that contain the desired palette values.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value The **_pg_setpalette** function returns 0 if there were no errors. If the new palettes are not valid, the value **_BADSCREENMODE** is returned.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_defaultchart, _pg_getpalette, _pg_initchart, _pg_resetpalette**

Example See the example for **_pg_getpalette.**

Description Sets the current styleset.

#include <pgchart.h>

void _far _pg_setstyleset(unsigned short _far ****styleset* **);**

styleset	Pointer to new styleset

Remarks The **_pg_setstyleset** function sets the current styleset.

Return Value None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_pg_defaultchart, _pg_getstyleset, _pg_initchart, _pg_resetstyleset**

Example See the example for **_pg_getpalette.**

Description Writes text vertically on the screen.

#include <pgchart.h>

short _far _pg_vlabelchart(chartenv _far **env*, **short** *x*, **short** *y*, **short** *color*,
char _far **label* **);**

env	Chart environment structure
x	Pixel *x* coordinate for text
y	Pixel *y* coordinate for text
color	Color code for text
label	Label text

Remarks The **_pg_vlabelchart** function writes text vertically on the screen. The arguments *x* and *y* are pixel coordinates for the beginning location of text relative to the upper-left corner of the chart window.

Return Value The **_pg_vlabelchart** function returns 0 if there were no errors. A nonzero value indicates a failure.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_pg_defaultchart**, **_pg_hlabelchart**, **_pg_initchart**

Example See the example for **_pg_getpalette**.

Description

Draw wedge-shaped figures.

#include <graph.h>

short _far _pie(short *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2*, **short** *x3*, **short** *y3*, **short** *x4*, **short** *y4* **);**

short _far _pie_w(short *control*, **double** *x1*, **double** *y1*, **double** *x2*, **double** *y2*, **double** *x3*, **double** *y3*, **double** *x4*, **double** *y4* **);**

short _far _pie_wxy(short *control*, **struct _wxycoord _far** **pwxy1*, **struct _wxycoord _far** **pwxy2*, **struct _wxycoord _far** **pwxy3*, **struct _wxycoord _far****pwxy4* **);**

control	Fill-control constant
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
x3, *y3*	Start vector
x4, *y4*	End vector
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
pwxy3	Start vector
pwxy4	End vector

Remarks

The **_pie** functions draw a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines.

The **_pie** function uses the view coordinate system. The center of the arc is the center of the bounding rectangle specified by the view coordinate points (*x1*, *y1*) and (*x2*, *y2*). The arc starts where it intersects the vector defined by (*x3*, *y3*) and ends where it intersects the vector (*x4*, *y4*).

The **_pie_wxy** and **_pie_w** functions use the window coordinate system. The center of the arc is the center of the bounding rectangle specified by the window coordinate pairs *pwxy1* and *pwxy2* for **_pie_wxy**, and by the points (*x1*, *y1*) and (*x2*, *y2*) for **_pie_w**. The arc starts where it intersects the vector defined by *pwxy3* or (*x3*, *y3*) and ends where it intersects the vector defined by *pwxy4* or (*x4*, *y4*).

The **_wxycoord** structure is defined in GRAPH.H and contains the following elements:

Element	Description
double wx	Window *x* coordinate
double wy	Window *y* coordinate

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

The control option given by **_GFILLINTERIOR** is equivalent to a subsequent call to the **_floodfill** function using the approximate center of the arc as the starting point and the current color (set by **_setcolor**) as the boundary color.

Return Value

These functions return a nonzero value if successful; otherwise, they return 0.

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

_arc functions, **_ellipse** functions, **_floodfill**, **_getcolor**, **_lineto** functions, **_rectangle** functions, **_setcolor**, **_setfillmask**

Example

```
/* PIE.C: This program draws a pie-shaped figure. */

#include <stdlib.h>
#include <conio.h>
#include <graph.h>
```

```
void main()
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _pie( _GBORDER, 80, 50, 240, 150, 240, 12, 0, 150 );
   getch();

   _setvideomode( _DEFAULTMODE );
}
```

Description

Creates a pipe for reading and writing.

#include <fcntl.h>	For O_BINARY and O_TEXT definitions
#include <errno.h>	**errno** definitions
#include <io.h>	Prototype declaration

int _pipe(int ****phandles*, unsigned int** *psize***, int** *textmode* **);**

phandles[2]	Array to hold read and write handles
psize	Amount of memory to reserve
textmode	File mode

Remarks

A pipe is an artificial file-like I/O channel that a program can create and use to pass information to other programs. A pipe is similar to a file in that it has a file pointer or a file descriptor, or both, and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead, a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the operating system.

Pipes may be used to pass information between programs. For example, the command processor in OS/2 creates a pipe when executing a command such as

```
PROGRAM1 | PROGRAM2
```

The standard output handle of PROGRAM1 is attached to the pipe's write handle. The standard input handle of PROGRAM2 is attached to the pipe's read handle. This eliminates the need for creating temporary files to pass information to other programs.

The **_pipe** function creates a pipe. This function is similar to **open** but opens the pipe for both reading and writing, returning two file handles instead of one. The program can either use both sides of the pipe or close the one it does not need. This function typically opens a pipe in preparation for linking it to a child process.

The **_pipe** function opens a pipe and returns two handles to the pipe in the *phandles* argument. The element *phandles*[0] contains the read handle, and the element *phandles*[1] contains the write handle. Pipe file handles are used in the same way as other file handles. (The low-level input and output functions **read** and **write** can read from and write to a pipe.)

The *psize* argument specifies the amount of memory, in bytes, to reserve for the pipe.

The *textmode* argument specifies the translation mode for the pipe. The manifest constant **O_TEXT** specifies a text translation, and the constant **O_BINARY** specifies binary translation. (See **fopen** for a description of text and binary modes.) If the *textmode* argument is 0, the **_pipe** function uses the default translation mode specified by the default-mode variable **_fmode**.

In multithread programs, no locking is performed. The handles returned are newly opened and should not be referenced by any thread until after the **_pipe** call is complete.

Under OS/2, a pipe is destroyed when all its handles have been closed. (If all read handles on the pipe have been closed, writing to the pipe will cause an error.) All read and write operations on the pipe wait until there is enough data or enough buffer space to complete the I/O request.

Return Value

The **_pipe** function returns 0 if successful. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
EMFILE	No more file handles available (too many open files)
ENFILE	System file table overflow

Compatibility

□ ANSI □ DOS ■ OS/2 ■ UNIX ■ XENIX

A similar function (**pipe**) is available in the XENIX and UNIX operating environments.

See Also

cwait, _pclose, _popen

Example

```
/* PIPE.C: This program uses _pipe to pass streams of text to
 * child processes.
 */

#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>          /* _pipe */
#include <math.h>
```

```
enum PIPES { READ, WRITE };     /* Constants 0 and 1 for READ and WRITE */
#define NUMPROBLEM 8

void main( int argc, char *argv[] )
{
    int     hpipe[2];
    char    hstr[20];
    int     termstat, pid, problem, c;

    /* If no arguments, this is the parent. */
    if( argc == 1 )
    {
        /* Open a sets of pipes. */
        if( _pipe( hpipe, 256, O_BINARY ) == -1 )
            exit( 1 );

        /* Convert pipe read handle to string and pass as argument to
         * spawned child. Program spawns itself (argv[0]).
         */
        itoa( hpipe[READ],  hstr,  10 );
        if( spawnl( P_NOWAIT, argv[0], argv[0], hstr, NULL )  == -1 )
            printf( "Spawn failed" );

        /* Put problem in write pipe. Since child is running simultaneously,
         * first solutions may be done before last problem is given.
         */
        for( problem = 1000; problem <= NUMPROBLEM * 1000; problem += 1000 )
        {
            printf( "Son, what is the square root of %d?\n", problem );
            write( hpipe[WRITE], (char *)&problem, sizeof( int ) );
        }

        /* Wait until child is done processing. */
        wait( &termstat );
        if( termstat & 0xff )
            printf( "Child failed\n" );

        close( hpipe[READ] );
        close( hpipe[WRITE] );
    }
```

```
    /* If there is an argument, this must be the child. */
    else
    {
        /* Convert passed string handle to integer handle. */
        hpipe[READ] = atoi( argv[1] );

        /* Read problem from pipe and calculate solution. */
        for( c = 0; c < NUMPROBLEM; c++ )
        {
            read( hpipe[READ], (char *)&problem, sizeof( int ) );
            printf( "Dad, the square root of %d is %3.2f.\n",
                    problem, sqrt( (double)problem ) );;
        }
    }
}
```

Output

```
Son, what is the square root of 1000?
Dad, the square root of 1000 is 31.62.
Son, what is the square root of 2000?
Son, what is the square root of 3000?
Dad, the square root of 2000 is 44.72.
Son, what is the square root of 4000?
Dad, the square root of 3000 is 54.77.
Son, what is the square root of 5000?
Dad, the square root of 4000 is 63.25.
Son, what is the square root of 6000?
Dad, the square root of 5000 is 70.71.
Son, what is the square root of 7000?
Dad, the square root of 6000 is 77.46.
Son, what is the square root of 8000?
Dad, the square root of 7000 is 83.67.
Dad, the square root of 8000 is 89.44.
```

Description Draw polygon shapes.

#include <graph.h>

short _far _polygon(short *control*, **struct xycoord _far** **points*, **short** *numpoints* **);**

short _far _polygon_w(short *control*, **double _far** **points*, **short** *numpoints* **);**

short _far _polygon_wxy(short *control*, **struct _wxycoord _far** **points*,
short *numpoints* **);**

control	Fill flag
points	Pointer to an array of structures defining the polygon
numpoints	Number of points

Remarks The **_polygon** functions draw polygons. The border of the polygon is drawn in the current color and line style. The **_polygon** routine uses the view coordinate system (expressed in **xycoord** structures), and the **_polygon_wxy** and **_polygon_w** routines use real-valued window coordinates (expressed in **_wxycoord** structures and in pairs of double-precision floating-point values, respectively).

The argument *points* is an array of **xycoord** or **_wxycoord** structures or pairs of doubles, each of which specifies one of the polygon's vertices. (For **_polygon_w**, *points*[0] and *points*[1] specify the *x* and *y* coordinates, respectively, of the first point.) The argument *numpoints* indicates the number of elements (the number of vertices) in the *points* array.

The *control* argument can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the polygon using the current fill mask
_GBORDER	Does not fill the polygon

The **_setwritemode**, **_setlinestyle**, and **_setfillmask** functions all affect the output from the **_polygon** functions.

Return Value The **_polygon** functions return a nonzero value if the arc is successfully drawn; otherwise, they return 0.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_ellipse** functions, **_floodfill**, **_lineto** functions, **_pie** functions, **_rectangle** functions, **_setcolor**, **_setfillmask**, **_setlinestyle**, **_setwritemode**

Example

```
/* POLYGON.C: This program draws a star-shaped polygon. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <math.h>
#include <stdlib.h>

#define PI 3.1415

void main()
{
   short  side, radius = 90, x = 0, y = 0;
   double radians;
   struct xycoord polyside[5];
   struct videoconfig vc;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _getvideoconfig( &vc );
   _setvieworg( vc.numxpixels / 2, vc. numypixels / 2 );

   /* Calculate points of star every 144 degrees, then connect them. */
   for( side = 0; side < 5; side++ )
   {
       radians = 144 * PI / 180;
       polyside[side].xcoord = x + (short)(cos( side * radians ) * radius);
       polyside[side].ycoord = y + (short)(sin( side * radians ) * radius);
   }
   _polygon( _GFILLINTERIOR, polyside, 5 );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description

Creates a pipe and executes a command.

#include <stdio.h>	Required for function declarations only

FILE *_popen(char ***_command_, char ***_mode_ **);**

command	Command to be executed
mode	Mode of returned stream

Remarks

The **_popen** function creates a pipe and asynchronously executes a child copy of the command processor with the specified command string *command.* See **_pipe** for a general discussion of pipes in OS/2. The character string *mode* specifies the type of access requested, as follows:

Type	Description
"r"	The calling process can read the child command's standard output via the returned stream.
"w"	The calling process can write to the child command's standard input via the returned stream.
"b"	Open in binary mode.
"t"	Open in text mode.

See Section 2.7, "Input and Output," for a discussion of text and binary modes.

Return Value

The **_popen** function returns a stream associated with one end of the created pipe. The other end of the pipe is associated with the child command's standard input or standard output. If an error occurs, **NULL** is returned.

Compatibility

☐ ANSI ☐ DOS ■ OS/2 ■ UNIX ■ XENIX

A similar function (**popen**) is available in the XENIX and UNIX operating environments.

See Also

_pclose, _pipe

Example

```
/* POPEN.C: This program uses _popen and _pclose to receive a stream
 * of text from a child system process.
 */
```

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

void main()
{
   char    buffer[128];
   FILE    *chkdsk;

   /* Run CHKDSK so that it writes its output to a pipe. Open pipe
    * with read text attribute so that we can read it like a text file.
    */
   if( (chkdsk = _popen( "dir po*.c | sort | more", "rt" )) == NULL )
      exit( 1 );

   /* Read pipe until end of file. End of file indicates that CHKDSK
    * closed its standard out (probably meaning it terminated).
    */
   while( !feof( chkdsk ) )
   {
      if( fgets( buffer, 128, chkdsk ) != NULL )
         printf( buffer );
   }

   /* Close pipe and print return value of CHKDSK. */
   printf( "\nChild returned %d\n", _pclose( chkdsk ) );
}
```

Output

```
     3 File(s)   12683264 bytes free
 Directory of C:\LIBREF
 The volume label in drive C is OS2.
POLYGON  C        921   6-14-89   6:51p
POPEN    C        845   6-19-89   2:48p
POW      C        190   6-13-89   6:07p

Child returned 0
```

Description

Calculate *x* raised to the power of *y*.

#include <math.h>

double pow(double *x*, double *y*);

long double powl(long double *x*, long double *y*);

x	Number to be raised
y	Power of *x*

Remarks

The **pow** and **powl** functions compute *x* raised to the power of *y*.

The **powl** function is the 80-bit counterpart, and it uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

The **pow** and **powl** functions return the value of x^y. If *x* is not 0.0 and *y* is 0.0, **pow** and **powl** return the value 1. If *x* is 0.0 and *y* is negative, **pow** and **powl** set **errno** to **EDOM** and return 0.0. If both *x* and *y* are 0.0, 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 0.0. 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`.

Compatibility

pow

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

powl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

exp, log functions, **sqrt**

Example

```
/* POW.C */
#include <math.h>
#include <stdio.h>
```

```
void main()
{
   double x = 2.Ø, y = 3.Ø, z;

   z = pow( x, y );
   printf( "%.1f to the power of %.1f is %.1f\n", x, y, z );
}
```

Output

```
2.Ø to the power of 3.Ø is 8.Ø
```

Description

Prints formatted output to the standard output stream.

#include <stdio.h>

int printf(const char ***format*** **[[, *argument*]]...);**

format	Format control
argument	Optional arguments

Remarks

The **printf** function formats and prints a series of characters and values to the standard output stream, **stdout**. The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are 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
```

If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments.

Format specifications always begin with a percent sign (%) and are read left to right. When the first format specification (if any) is encountered, the value of the first argument after *format* is converted and output accordingly. 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, which consists of optional and required fields, has the following form:

%[[*flags*]] [[*width*]] [[.*precision*]] [[{**F** | **N** | **h** | **l** | **L**}]]*type*

Format Specification Fields

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`). The optional fields, which appear before the *type* character, control other aspects of the formatting. The fields in a **printf** format specification are described in the following list:

Field	Description
type	Required character that determines whether the associated argument is interpreted as a character, a string, or a number. (See Table R.2.)
flags	Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. (See Table R.3.) More than one flag can appear in a format specification.
width	Optional number that specifies minimum number of characters output.
precision	Optional number that specifies 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.4.)
F, N	Optional prefixes that refer to the "distance" to the object being printed (**near** or **far**). F and N are not part of the ANSI definition for **printf.** They are Microsoft extensions that should not be used if ANSI portability is desired.
h, l, L	Optional prefixes that determine the size of the argument expected, as shown below:

Prefix	Use
h	Used with the integer types **d**, **i**, **o**, **x**, and **X** to specify that the argument is **short int**, or with **u** to specify **short unsigned int.** If used with **%p**, it indicates a 16-bit pointer.
l	Used with **d**, **i**, **o**, **x**, and **X** types to specify that the argument is **long int**, or with **u** to specify **long unsigned int**; also used with **e**, **E**, **f**, **g**, and **G** types to specify **double** rather than **float.** If used with **%p,** it indicates a 32-bit pointer.
L	Used with **e**, **E**, **f**, **g**, and **G** types to specify **long double.**

If a percent sign is followed by a character that has no meaning as a format field, the character is copied to **stdout.** For example, to print a percent-sign character, use %%.

Type Field Characters

The *type* character is the only required format field for the **printf** function; it appears after any optional format fields. The *type* character determines whether the associated argument is interpreted as a character, a string, or a number. (See Table R.2.)

Table R.2 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**, rather than **e**, introduces the exponent.
g	**double**	Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision. The **e** format is used only when the exponent of the value is less than –4 or greater than or equal to 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**, rather than **g**, introduces the exponent (where appropriate).
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; **%hp** indicates a near pointer and prints only the offset of the address.

Flag Directives

The first optional field of the format specification is *flag*. A flag directive is a character that justifies output and prints signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification. (See Table R.3.)

Table R.3 Flag Characters for printf

Flag	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 (–).
0	If *width* is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (**i, u, x, X, o, d**), the 0 is ignored.	No padding.
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 appears.
#	When used with the o, **x**, or **X** format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
	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.	

Width Specification

The second optional field of the format specification is the width specification. 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 (for left justification) is specified—until the minimum width is reached. If *width* is prefixed with 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.

Precision Specification

The third optional field of the format specification is the precision specification. It specifies 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.4.) Unlike the width specification, the precision specification can cause truncation of the output value, or rounding in the case of a floating-point value. If *precision* is specified as zero and the value to be converted is zero, the result is no characters output, as shown below:

```
printf( "%.0d", 0 );      /* No characters output */
```

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.4.

Table R.4 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 6; if *precision* is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g **G**	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, with any trailing zeros truncated.
c	The precision has no effect.	Character is 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.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or not a number (NAN), the **printf** function gives the following output:

Value	Output
+ infinity	**1.#INF***random-digits*
– infinity	**–1.#INF***random-digits*
Indefinite	*digit***.#IND***random-digits*
NAN	*digit***.#NAN***random-digits*

Size and Distance Specification

For **printf**, the format specification fields **F** and **N** refer to the "distance" to the object being read (**near** or **far**), and **h** and **l** refer to the "size" of the object being read (16-bit **short** or 32-bit **long**). The following list clarifies this use of **F**, **N**, **h**, **l**, and **L**:

Program Code	Action
printf ("%Ns");	Print **near string**
printf ("%Fs");	Print **far string**
printf ("%Nn");	Store **char** count in **near int**
printf ("%Fn");	Store **char** count in **far int**
printf ("%hp");	Print a 16-bit pointer (*xxxx*)
printf ("%lp");	Print a 32-bit pointer (*xxxx:xxxx*)
printf ("%Nhn");	Store **char** count in **near short int**
printf ("%Nln");	Store **char** count in **near long int**
printf ("%Fhn");	Store **char** count in **far short int**
printf ("%Fln");	Store **char** count in **far int**

The specifications **"%hs"** and **"%ls"** are meaningless to **printf**. The specifications **"%Np"** and **"%Fp"** are aliases for **"%hp"** and **"%lp"** for the sake of compatibility with Microsoft C version 4.0.

Return Value

The **printf** function returns the number of characters printed, or a negative value in the case of an error.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also fprintf, scanf, sprintf, vfprintf, vprintf, vsprintf

Example

```
/* PRINTF.C illustrates output formatting with printf. */

#include <stdio.h>

void main()
{
   char   ch = 'h', *string = "computer";
   int    count = -9234;
   double fp = 251.7366;

   /* Display integers. */
   printf( "Integer formats:\n"
           "\tDecimal: %d  Justified: %.6d  Unsigned: %u\n",
            count, count, count, count );

   printf( "Decimal %d as:\n\tHex: %Xh  C hex: 0x%x  Octal: %o\n",
            count, count, count, count );

   /* Display in different radixes. */
   printf( "Digits 10 equal:\n\tHex: %i  Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );

   /* Display characters. */
   printf( "Characters in field:\n%10c   %5c\n", ch, ch );

   /* Display strings. */
   printf( "Strings in field:\n%25s\n%25.4s\n", string, string );

   /* Display real numbers. */
   printf( "Real numbers:\n\t%f    %.2f    %e    %E\n", fp, fp, fp, fp );

   /* Display pointers. */
   printf( "Address as:\n\tDefault: %p  Near: %Np  Far: %Fp\n",
            &count, (int _near *)&count, (int _far *)&count );

   /* Count characters printed. */
   printf( "Display to here:\n" );
   printf( "1234567890123456%n78901234567890\n", &count );
   printf( "\tNumber displayed: %d\n\n", count );

}
```

Output

```
Integer formats:
        Decimal: -9234  Justified: -009234  Unsigned: 56302
Decimal -9234 as:
        Hex: DBEEh  C hex: 0xdbee  Octal: 155756
Digits 10 equal:
        Hex: 16  Octal: 8  Decimal: 10
Characters in field:
         h       h
Strings in field:
                 computer
                     comp
Real numbers:
        251.736600    251.74    2.517366e+002    2.517366E+002
Address as:
        Default: 141C  Near: 141C  Far: 0087:141C
Display to here:
123456789012345678901234567890
        Number displayed: 16
```

Description

Writes a character to a stream (**putc**) or to **stdout** (**putchar**).

#include <stdio.h>

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

int putchar(int *c* **);**

c	Character to be written
stream	Pointer to **FILE** structure

Remarks

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)**.

These routines are implemented as both macros and functions. See Section 1.4, "Choosing Between Functions and Macros," for a discussion of how to select between the macro and function forms.

Return Value

The **putc** and **putchar** routines return the character written, or **EOF** in the case of an error. Any integer can be passed to **putc**, but only the lower 8 bits are written.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fputc, fputchar, getc, getchar

Example

```
/* PUTC.C: This program uses putc to write buffer to a stream.
 * If an error occurs, the program will stop before writing the
 * entire buffer.
 */

#include <stdio.h>

void main()
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;
```

```
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putc( *p, stream );
}
```

Output

```
This is the line of output
```

Description

Writes a character to the console.

#include <conio.h>	Required only for function declarations

int putch(int *c*);

c	Character to be output

Remarks

The **putch** function writes the character *c* directly (without buffering) to the console.

Return Value

The function returns *c* if successful, and **EOF** if not.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

cprintf, getch, getche

Example

```
/* GETCH.C: This program reads characters from the keyboard until it
 * receives a 'Y' or 'y'.
 */

#include <conio.h>
#include <ctype.h>

void main()
{
   int ch;

   cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = getch();
      ch = toupper( ch );
   } while( ch != 'Y' );

   putch( ch );
   putch( '\r' );    /* Carriage return */
   putch( '\n' );    /* Line feed       */
}
```

Output

```
Type 'Y' when finished typing keys: Y
```

Description

Creates new environment variables.

#include <stdlib.h>	Required only for function declarations

int putenv(char **envstring* **);**

envstring	Environment-string definition

Remarks

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, its value is replaced by *string*; otherwise, the new *varname* variable and its *string* value are 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 modify the command-level environment. When the currently running process terminates, the environment reverts to the level of the parent process (in most cases, the operating system level). However, the environment affected by **putenv** can be passed to any child processes created by **spawn**, **exec**, **system**, or (in OS/2 only) **_popen**, and these child processes get any new items added by **putenv**.

Never free a pointer to an environment entry, because the environment variable will then point to freed space. A similar problem can occur if you pass **putenv** a pointer to a local variable, then exit the function in which the variable is declared.

The **putenv** function operates only on data structures accessible to the run-time library and not on the environment "segment" created for a process by DOS or OS/2.

Note that 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 is 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **getenv**

Example

```
/* GETENV.C: This program uses getenv to retrieve the LIB environment
 * variable and then uses putenv to change it to a new value.
 */

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

main()
{
   char *libvar;

   /* Get the value of the LIB environment variable. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );

   /* Attempt to change path. Note that this only affects the environment
    * variable of the current process. The command processor's environment
    * is not changed.
    */
   putenv( "LIB=c:\\mylib;c:\\yourlib" );

   /* Get new value. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );

}
```

Output

```
Original LIB variable is: C:\LIB
New LIB variable is: c:\mylib;c:\yourlib
```

Description

Retrieve images from a buffer.

#include <graph.h>

void _far _putimage(short *x*, **short** *y*, **char _huge** **image*, **short** *action* **);**

void _far _putimage_w(double *wx*, **double** *wy*, **char _huge** **image*, **short** *action* **);**

x, *y*	Position of upper-left corner of image
image	Stored image buffer
action	Interaction with existing screen image
wx, *wy*	Position of upper-left corner of image

Remarks

The **_putimage** function transfers to the screen the image stored in the buffer that *image* points to.

In the **_putimage** function, the upper-left corner of the image is placed at the view coordinate point (*x*, *y*). In the **_putimage_w** function, the upper-left corner of the image is placed at the window coordinate point (*wx*, *wy*).

The *action* argument defines the interaction between the stored image and the one that is 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 joined by logical-AND.
_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 None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getimage, _imagesize**

Example See the example for **_getimage.**

Description Writes a string to **stdout.**

#include <stdio.h>

int puts(const char ***string*);**

string	String to be output

Remarks The **puts** function writes *string* to the standard output stream **stdout**, replacing the string's terminating null character (**'\0'**) with a newline character (**\n**) in the output stream.

Return Value The **puts** function returns a nonnegative value if it is successful. If the function fails, it returns **EOF.**

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fputs, gets**

Example

```
/* PUTS.C: This program uses puts to write a string to stdout. */

#include <stdio.h>

void main()
{
   puts( "Hello world from puts!" );
}
```

Output

```
Hello world from puts!
```

Description Writes an integer to a stream.

#include <stdio.h>

int putw(int *binint*, **FILE** **stream* **);**

binint	Binary integer to be output
stream	Pointer to **FILE** structure

Remarks The **putw** function writes a binary value of type **int** to the current position of *stream*. The **putw** function does not affect the alignment of items in the stream, nor does it assume any special alignment.

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.

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.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **getw**

Example

```
/* PUTW.C: This program uses putw to write a word to a stream,
 * then performs an error check.
 */

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

void main()
{
   FILE *stream;
   unsigned u;

   if( (stream = fopen( "data.out", "wb" )) == NULL )
      exit( 1 );
   for( u = 0; u < 10; u++ )
   {
      putw( u + 0x2132, stdout );
      putw( u + 0x2132, stream );   /* Write word to stream. */
```

```
      if( ferror( stream ) )        /* Make error check. */
      {
         printf( "putw failed" );
         clearerr( stream );
         exit( 1 );
      }
   }
   printf( "\nWrote ten words\n" );
   fclose( stream );
}
```

Output

```
2!3!4!5!6!7!8!9!:!;!
Wrote ten words
```

Description

Performs a quick sort.

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

void qsort(void *****_base_**, size_t** _num_**, size_t** _width_**,**
int(***_compare_ **) (const void ***_elem1_, const void ***_elem2_ **));**

base	Start of target array
num	Array size in elements
width	Element size in bytes
compare	Comparison function
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

Remarks

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:

compare(**(void *)** *elem1*, **(void *)** *elem2*);

The routine must compare the elements, then return one of the following values:

Value	Meaning
< 0	*elem1* less than *elem2*
= 0	*elem1* equivalent to *elem2*
> 0	*elem1* greater than *elem2*

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

Return Value None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **bsearch, lsearch**

Example

```
/* QSORT.C: This program reads the command-line parameters and
 * uses qsort to sort them. It then displays the sorted arguments.
 */

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

int compare( char **arg1, char **arg2 );  /* Prototype */

void main( int argc, char **argv )
{
   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 ", argv[i] );
   printf( "\n" );
}

int compare( char **arg1, char **arg2 )
{
   /* Compare all of both strings: */
   return strcmpi( *arg1, *arg2 );
}
```

Output

```
[C:\LIBREF] qsort every good boy deserves favor
boy deserves every favor good
```

Description

Sends a signal to the executing program.

#include <signal.h>

int raise(int *sig* **);**

sig	Signal to be raised

Remarks

The **raise** function sends *sig* to the executing program. If a signal-handling routine for *sig* has ben installed by a prior call to **signal, raise** causes that routine to be executed. If no handler routine has been installed, the default action (as listed below) is taken.

The signal value *sig* can be one of the following manifest constants:

Signal	Meaning	Default
SIGABRT	Abnormal termination.	Terminates the calling program with exit code 3.
SIGBREAK	CTRL+ BREAK interrupt.	Terminates the calling program with exit code 3.
SIGFPE	Floating-point error.	Terminates the calling program.
SIGILL	Illegal instruction. This signal is not generated by DOS or OS/2, but is supported for ANSI compatibility.	Terminates the calling program.
SIGINT	CTRL+ C interrupt.	Issues INT23H.
SIGSEGV	Illegal storage access. This signal is not generated by DOS or OS/2, but is supported for ANSI compatiblity.	Terminates the calling program.
SIGTERM	Termination request sent to the program. This signal is not generated by DOS or OS/2, but is supported for ANSI compatibility.	Ignores the signal.
SIGUSR1 **SIGUSR2** **SIGUSR3**	User-defined signals.	Ignores the signal.

Return Value If successful, the **raise** function returns 0. Otherwise, it returns a nonzero value.

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **abort, signal**

Example See the example for **signal**.

Description

Generates a pseudorandom number.

#include <stdlib.h> Required only for function declarations

int rand(void);

Remarks

The **rand** function returns a pseudorandom integer in the range 0 to **RAND_MAX.** The **srand** routine can be used to seed the pseudorandom-number generator before calling **rand.**

Return Value

The **rand** function returns a pseudorandom number, as described above. There is no error return.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

srand

Example

```
/* RAND.C: This program seeds the random-number generator with the
 * time, then displays 20 random integers.
 */

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

void main()
{
   int i;

   /* Seed the random-number generator with current time so that
    * the numbers will be different every time we run.
    */
   srand( (unsigned)time( NULL ) );

   /* Display 10 numbers. */
   for( i = 0; i < 10; i++ )
       printf( "  %6d\n", rand() );
}
```

Output

```
19471
16395
 8268
15582
 6489
28356
27042
 5276
23070
10930
```

Description

Reads data from a file.

#include <io.h>	Required only for function declarations

int read(int *handle*, **void** **buffer*, **unsigned int** *count* **);**

handle	Handle referring to open file
buffer	Storage location for data
count	Maximum number of bytes

Remarks

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 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 (DOS versions 3.0 and later and OS/2 only) the file is locked.

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 in one operation is 65,534, since 65,535 (or 0xFFFF) is indistinguishable from –1, and therefore cannot be distinguished from an error return.

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 (CR-LF) 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.

Note that under DOS and OS/2, when files are opened in text mode, a CTRL+Z 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 **lseek** function will clear the end-of-file indicator.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also creat, fread, open, write

Example

```
/* READ.C: This program opens a file named READ.C and tries to read 60,000
 * bytes from that file using read. It then displays the actual
 * number of bytes read from READ.C.
 */

#include <fcntl.h>      /* Needed only for O_RDWR definition */
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

char buffer[60000];

void main()
{
   int fh;
   unsigned int nbytes = 60000, bytesread;

   /* Open file for input: */
   if( (fh = open( "read.c", 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 );

   close( fh );
}
```

Output

```
Read 747 bytes from file
```

Description

Reallocate memory blocks.

#include <stdlib.h>	For ANSI compatibility (**realloc** only)
#include <malloc.h>	Required only for function declarations

void *realloc(void ****memblock*, size_t** *size* **);**

void _based(void) *_brealloc(_segment *seg***, void _based(void) ****memblock***,**
size_t *size* **);**

void _far *_frealloc(void _far ****memblock*, size_t** *size* **);**

void _near *_nrealloc(void _near ****memblock*, size_t** *size* **);**

memblock	Pointer to previously allocated memory block
size	New size in bytes
seg	Segment selector

Remarks

The **realloc** family of functions changes the size of a previously allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is **NULL**, **realloc** functions in the same way as **malloc** and allocates a new block of *size* bytes. If *memblock* 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, although the new block may be in a different location.

The *memblock* argument can also point to a block that has been freed, as long as there has been no intervening call to the corresponding **calloc**, **malloc**, **_expand**, or **realloc** function. If successful, the reallocated block is marked in use.

In large data models (that is, compact-, large-, and huge-model programs), **realloc** maps to **_frealloc**. In small data models (tiny-, small-, and medium-model programs), **realloc** maps to **_nrealloc**.

The various **realloc** functions reallocate memory in the heap specified in the following list:

Function	Heap
realloc	Depends on data model of program
_brealloc	Based heap specified by *seg* value
_frealloc	Far heap (outside default data segment)
_nrealloc	Near heap (inside default data segment)

Return Value

The **realloc** functions return a **void** pointer to the reallocated (and possibly moved) memory block.

The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

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.

Compatibility

realloc

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_brealloc, _frealloc, _nrealloc

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

calloc functions, **free** functions, **malloc** functions

Example

```
/* REALLOC.C: This program allocates a block of memory for buffer
 * and then uses _msize to display the size of that block. Next, it
 * uses realloc to expand the amount of memory used by buffer
 * and then calls _msize again to display the new amount of
 * memory allocated to buffer.
 */
```

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

void main()
{
   long *buffer;
   size_t size;

   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );

   size = _msize( buffer );
   printf( "Size of block after malloc of 1000 longs: %u\n", size );

   /* Reallocate and show new size: */
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) )) == NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after realloc of 1000 more longs: %u\n", size );

   free( buffer );
}
```

Output

```
Size of block after malloc of 1000 longs: 4000
Size of block after realloc of 1000 more longs: 8000
```

Description

Draw rectangles.

#include <graph.h>

short _far _rectangle(short *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

short _far _rectangle_w(short *control*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

short _far _rectangle_wxy(short *control*, **struct _wxycoord _far** **pwxy1*, **struct _wxycoord _far** **pwxy2* **);**

control	Fill flag
x1, *y1*	Upper-left corner
x2, *y2*	Lower-right corner
wx1, *wy1*	Upper-left corner
wx2, *wy2*	Lower-right corner
pwxy1	Upper-left corner
pwxy2	Lower-right corner

Remarks

The **_rectangle** functions draw a rectangle with the current line style.

The **_rectangle** function uses the view coordinate system. The view coordinate points (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle.

The **_rectangle_w** function uses the window coordinate system. The window coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*) are the diagonally opposed corners of the rectangle.

The **_rectangle_wxy** function uses the window coordinate system. The window coordinate points (*pwxy1*) and (*pwxy2*) are the diagonally opposed corners of the rectangle. The coordinates for the **_rectangle_wxy** routine are given in terms of an **_wxycoord** structure (defined in GRAPH.H), which contains the following elements:

Element	Description
double wx	window *x* coordinate
double wy	window *y* coordinate

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.

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 if not.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_arc functions, **_ellipse** functions, **_floodfill**, **_getcolor**, **_lineto** functions, **_pie** functions, **_setcolor**, **_setfillmask**

Example

```
/* RECT.C: This program draws a rectangle. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _rectangle( _GBORDER, 80, 50, 240, 150 );

   getch();

   _setvideomode( _DEFAULTMODE );
}
```

Description

Initializes the fonts graphics system.

#include <graph.h>

short _far _registerfonts(unsigned char _far ***pathname*** **);**

pathname	Path name specifying .FON files to be registered

Remarks

The **_registerfonts** function initializes the fonts graphics system. Font files must be registered with the **_registerfonts** function before any other font-related library function (**_getgtextextent, _outgtext, _setfont, _unregisterfonts**) can be used.

The **_registerfonts** function reads the specified files and loads font header information into memory. Each font header takes up about 140 bytes of memory.

The *pathname* argument is the path specification and file name of valid .FON files. The *pathname* can contain standard DOS wild-card characters.

The font functions affect only the output from the font output function **_outgtext**; no other C run-time output functions are affected by font usage.

Return Value

The **_registerfonts** function returns a positive value which indicates the number of fonts successfully registered. A negative return value indicates failure. The following negative values may be returned:

Value	Meaning
–1	No such file or directory.
–2	One or more of the .FON files was not a valid, binary .FON file.
–3	One or more of the .FON files is damaged.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getfontinfo, _getgtextextent, _outgtext, _setfont, _unregisterfonts

Example

See the example for **_outgtext**.

Description

Remap all palette colors.

#include <graph.h>

short _far _remapallpalette(long _far **colors* **);**

long _far _remappalette(short *index*, **long** *color* **);**

colors	Color value array
index	Color index to reassign
color	Color value to assign color index to

Remarks

The **_remapallpalette** function remaps the entire color palette simultaneously to the colors given in the *colors* array. The *colors* array is an array of **long** integers where the size of the array varies from 16 to 64 to 256, depending on the video mode. The number of colors mapped depends on the number of colors supported by the current video mode. The **_remapallpalette** function works in all video modes (except **_ORESCOLOR** mode), but only with EGA, MCGA, or VGA hardware.

The default color values for a color text on 16-color graphics mode are shown below:

Number	Color	Number	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 first array element specifies the new color value to be associated with color index 0 (the background color in graphics modes). After the call to **_remapallpalette**, calls to **_setcolor** will index into the new array of colors. The mapping done by **_remapallpalette** affects the current display immediately.

The *colors* array can be larger than the number of colors supported by the current video mode, but only the first *n* elements are used, where *n* is the number of colors supported by the current video mode, as indicated by the **numcolors** element of the **videoconfig** structure.

The **long** color value is defined by specifying three bytes of data representing the three component colors: red, green, and blue.

Each of the three bytes represents the intensity of one of the red, green, or blue component colors, and must be in the range 0–31. In other words, the low-order six bits of each byte specify the component's intensity and the high-order two bits should be zero. The fourth (high-order) byte in the **long** is unused and should be set to zero. The diagram below shows the ordering of bytes within the **long** value.

For example, to create a lighter shade of blue, start with lots of blue, add some green, and maybe a little bit of red. The three-byte color value would be:

```
blue byte          green byte        red byte
00011111           00101111          00011111
high ------------> low order
```

Manifest constants are defined in GRAPH.H for the default color values corresponding to color indices 0–15 in color text modes and 16-color graphics modes, as shown below:

Index	Constant	Index	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	_YELLOW
7	_WHITE	15	_BRIGHTWHITE

The VGA supports a palette of 262,144 colors (256K) in color modes, and the EGA supports a palette of only 64 different colors. Color values for EGA are specified in exactly the same way as with the VGA; however, the low-order four bits of each byte are simply ignored.

The **_remappalette** function assigns a new color value *color* to the color index given by *index*. This remapping affects the current display immediately.

The **_remappalette** function works in all graphics modes, but only with EGA, MCGA, or VGA hardware. An error results if the function is called while using any other configuration.

The color value used in **_remappalette** is defined and used exactly as noted above for **_remapallpalette**. The range of color indices used with **_remappalette** depends on the number of colors supported by the video mode.

The **_remapallpalette** and **_remappalette** functions do not affect the presentation-graphics palettes, which are manipulated with the **_pg_getpalette**, **_pg_setpalette**, and **_pg_resetpalette** functions.

If a VGA or MCGA adapter is connected to an analog monochrome monitor, the color value is transformed into its gray-scale equivalent, based on the weighted sum of its red, green, and blue components (30% red + 50% green + 11% blue). The original red, green, and blue values are lost.

Return Value

If successful, **_remapallpalette** returns –1 (short). In case of an error, **_remapallpalette** returns 0 (short).

If successful, **_remappalette** returns the color value previously assigned to *index*, or –1 if the function is inoperative (not EGA, VGA, or MCGA), or if the color index is out of range.

Note that **_remapallpalette** returns a **short** value and **_remappalette** returns a **long** value.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_selectpalette, _setbkcolor, _setvideomode

Example

```
/* RMPALPAL.C: This example illustrates functions for assigning
 * color values to color indices. Functions illustrated include:
 *    _remappalette          _remapallpalette
 */

#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

/* Macro for mixing Red, Green, and Blue elements of color */
#define RGB(r,g,b) (((long) ((b) << 8 | (g)) << 8) | (r))
```

```
long tmp, pal[256];
void main()
{
   short  red, blue, green;
   short  inc, i, mode, cells, x, y, xinc, yinc;
   char   buf[40];
   struct videoconfig vc;

   /* Make sure all palette numbers are valid. */
   for( i = 0; i < 256; i++ )
      pal[i] = _BLACK;

   /* Loop through each graphics mode that supports palettes. */
   for( mode = _MRES4COLOR; mode <= _MRES256COLOR; mode++ )
   {
      if( mode == _ERESNOCOLOR )
         mode++;
      if( !_setvideomode( mode ) )
         continue;

      /* Set variables for each mode. */
      _getvideoconfig( &vc );
      switch( vc.numcolors )
      {
         case 256:          /* Active bits in this order:           */
            cells = 13;
            inc = 12;       /* ???????? ??bbbbbb ??gggggg ??rrrrrr  */
            break;
         case  16:
            cells = 4;
            if( (vc.mode == _ERESCOLOR) || (vc.mode == _VRES16COLOR) )
               inc = 16;    /* ???????? ??bb???? ??gg???? ??rr????  */
            else
               inc = 32;    /* ???????? ??Bb???? ??Gg???? ??Rr????  */
            break;
         case   4:
            cells = 2;
            inc = 32;       /* ???????? ??Bb???? ??Gg???? ??Rr????  */
            break;
         default:
             continue;
      }
      xinc = vc.numxpixels / cells;
      yinc = vc.numypixels / cells;
```

```
/* Fill palette arrays in BGR order. */
for( i = 0, blue = 0; blue < 64; blue += inc )
   for( green = 0; green < 64; green += inc )
      for( red = 0; red < 64; red += inc )
      {
         pal[i] = RGB( red, green, blue );
         /* Special case of using 6 bits to represent 16 colors.
          * If both bits are on for any color, intensity is set.
          * If one bit is set for a color, the color is on.
          */
         if( inc == 32 )
            pal[i + 8] = pal[i] | (pal[i] >> 1);
         i++;
      }

/* If palettes available, remap all palettes at once. */
if( !_remapallpalette( pal ) )
{
   _setvideomode( _DEFAULTMODE );
   _outtext( "Palettes not available with this adapter" );
   exit( 1 );
}

/* Draw colored squares. */
for( i = 0, x = 0; x < ( xinc * cells ); x += xinc )
   for( y = 0; y < ( yinc * cells); y += yinc )
   {
      _setcolor( i++ );
      _rectangle( _GFILLINTERIOR, x, y, x + xinc, y + yinc );
   }

/* Note that for 256-color mode, not all colors are shown. The number
 * of colors from mixing three base colors can never be the same as
 * the number that can be shown on a two-dimensional grid.
 */
sprintf( buf, "Mode %d has %d colors", vc.mode, vc.numcolors );
_setcolor( vc.numcolors / 2 );
_outtext( buf );
getch();
```

```
      /* Change each palette entry separately in GRB order. */
      for( i = Ø, green = Ø; green < 64; green += inc )
         for( red = Ø; red < 64; red += inc )
            for(blue = Ø; blue < 64; blue += inc )
            {
               tmp = RGB( red, green, blue );
               _remappalette( i, tmp );
               if( inc == 32 )
                  _remappalette( i + 8, tmp | (tmp >> 1) );
               i++;
            }
      getch();
   }
   _setvideomode( _DEFAULTMODE );
}
```

Description

Deletes a file.

#include <stdio.h>	Required for ANSI compatibility
#include <io.h>	Use either IO.H or STDIO.H

int remove(const char ***path*** **);**

path	Path name of file to be removed

Remarks

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 read-only file.
ENOENT	File or path name not found, or path name specifies a directory.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

unlink

Example

```
/* REMOVE.C: This program uses remove to delete REMOVE.OBJ. */

#include <stdio.h>

void main()
{
   if( remove( "remove.obj" ) == -1 )
      perror( "Could not delete 'REMOVE.OBJ'" );
   else
      printf( "Deleted 'REMOVE.OBJ'\n" );
}
```

Output

```
Deleted 'REMOVE.OBJ'
```

Description Renames a file or directory.

#include <stdio.h>	Required for ANSI compatibility
#include <io.h>	Use either IO.H or STDIO.H

int rename(const char ***oldname*, const char** ***newname*);**

oldname	Pointer to old name
newname	Pointer to new name

Remarks 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

Example

```
/* RENAMER.C: This program attempts to rename a file named RENAMER.OBJ to
 * RENAMER.JBO. For this operation to succeed, a file named RENAMER.OBJ
 * must exist and a file named RENAMER.JBO must not exist.
 */

#include <stdio.h>
```

```
void main()
{
   int  result;
   char old[] = "RENAMER.OBJ", new[] = "RENAMER.JBO";

   /* Attempt to rename file: */
   result = rename( old, new );
   if( result != 0 )
      printf( "Could not rename '%s'\n", old );
   else
      printf( "File '%s' renamed to '%s'\n", old, new );
}
```

Output

```
File 'RENAMER.OBJ' renamed to 'RENAMER.JBO'
```

Description

Repositions the file pointer to the beginning of a file.

#include <stdio.h>

void rewind(FILE ****stream*** **);**

stream	Pointer to **FILE** structure

Remarks

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 error indicators for the stream, and **fseek** does not. Both **rewind** and **fseek** clear the end-of-file indicator. Also, **fseek** returns a value that indicates whether the pointer was successfully moved, but **rewind** does not return any value.

You can also use the **rewind** function to clear the keyboard buffer. Use the **rewind** function with the stream **stdin**, which is associated with the keyboard by default.

Return Value

The **rewind** function has no return value.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

Example

```
/* REWIND.C: This program first opens a file named REWIND.OUT 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.
 */

#include <stdio.h>

void main()
{
   FILE *stream;
   int data1, data2;

   data1 = 1;
   data2 = -37;

   if( (stream = fopen( "rewind.out", "w+" )) != NULL )
   {
      fprintf( stream, "%d %d", data1, data2 );
      printf( "The values written are: %d and %d\n", data1, data2 );
```

```
        rewind( stream );
        fscanf( stream, "%d %d", &data1, &data2 );
        printf( "The values read are: %d and %d\n", data1, data2 );
        fclose( stream );
    }
}
```

Output

```
The values written are: 1 and -37
The values read are: 1 and -37
```

Description

Deletes a directory.

#include <direct.h> Required only for function declarations

int rmdir(char ****dirname* **);**

dirname Path name of directory to be removed

Remarks

The **rmdir** function deletes the directory specified by *dirname*. 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 the root directory.
ENOENT	Path name not found.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

chdir, mkdir

Example

```
/* MAKEDIR.C */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

void main()
{
   int result;
```

```
   if( mkdir( "\\testtmp" ) == 0 )
   {
      printf( "Directory '\\testtmp' was successfully created\n" );
      system( "dir \\testtmp" );
      if( rmdir( "\\testtmp" ) == 0 )
         printf( "Directory '\\testtmp' was successfully removed\n"  );
      else
         printf( "Problem removing directory '\\testtmp'\n" );
   }
   else
      printf( "Problem creating directory '\\testtmp'\n" );
}
```

Output

```
Directory '\testtmp' was successfully created

 The volume label in drive C is OS2.
 Directory of C:\TESTTMP

.            <DIR>      6-19-89  11:20a
..           <DIR>      6-19-89  11:20a
     2 File(s)   12730368 bytes free
Directory '\testtmp' was successfully removed
```

Description Removes temporary files.

#include <stdio.h>

int rmtmp(void);

Remarks 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.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **flushall, tmpfile, tmpnam**

Example

```
/* TMPFILE.C: This program uses tmpfile to create a temporary file,
 * then deletes this file with rmtmp.
 */

#include <stdio.h>

void main()
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;

   /* Create temporary files. */
   for( i = 1; i <= 10; i++ )
   {
      if( (stream = tmpfile()) == NULL )
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }

   /* Remove temporary files. */
   printf( "%d temporary files deleted\n", rmtmp() );
}
```

Output

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
Temporary file 4 was created
Temporary file 5 was created
Temporary file 6 was created
Temporary file 7 was created
Temporary file 8 was created
Temporary file 9 was created
Temporary file 10 was created
10 temporary files deleted
```

Description Rotate bits to the left (**_rotl**) or right (**_rotr**).

#include <stdlib.h>

unsigned int _rotl(unsigned int *value*, **int** *shift* **);**

unsigned int _rotr(unsigned int *value*, **int** *shift* **);**

value	Value to be rotated
shift	Number of bits to shift

Remarks The **_rotl** and **_rotr** functions rotate the **unsigned** *value* by *shift* bits. The **_rotl** function rotates the value left. The **_rotr** function rotates the value right. Both functions "wrap" bits rotated off one end of *value* to the other end.

Return Value Both functions return the rotated value. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_lrotl**, **_lrotr**

Example

```
/* ROT.C: This program uses _rotr and _rotl with different shift
 * values to rotate an integer.
 */

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

void main()
{
   unsigned val = 0x0fd93;

   printf( "0x%4.4x rotated left three times is 0x%4.4x\n",
           val, _rotl( val, 3 ) );
   printf( "0x%4.4x rotated right four times is 0x%4.4x\n",
           val, _rotr( val, 4 ) );
}
```

Output

```
0xfd93 rotated left three times is 0xec9f
0xfd93 rotated right four times is 0x3fd9
```

Description

Reads formatted data from the standard input stream.

#include <stdio.h>

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

format	Format control
argument	Optional argument

Remarks

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 newline (**\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 matching characters in **stdin** are scanned but not stored. If a character in **stdin** conflicts with the format specification, **scanf** terminates. The 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 this specification and stored in the location that is 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 newline), or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached. If there are too many arguments for the given specifications, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments for the format specification.

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 newline) 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 specify whether the argument is far or near, respectively. **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.

For **scanf**, **N** and **F** refer to the "distance" to the object being read in (near or far) and **h** and **l** refer to the "size" of the object being read in (16-bit short or 32-bit long). The list below clarifies this use of **N**, **F**, **l**, and **h**:

Program Code	Action
scanf("%Ns", &x);	Read a string into near memory
scanf("%Fs", &x);	Read a string into far memory
scanf("%Nd", &x);	Read an **int** into near memory
scanf("%Fd", &x);	Read an **int** into far memory
scanf("%Nld", &x);	Read a **long int** into near memory
scanf("%Fld", &x);	Read a **long int** into far memory
scanf("%Nhp", &x);	Read a 16-bit pointer into near memory
scanf("%Nlp", &x);	Read a 32-bit pointer into near memory
scanf("%Fhp", &x);	Read a 16-bit pointer into far memory
scanf("%Flp", &x);	Read a 32-bit pointer into far memory

The type characters and their meanings are described in Table R.5.

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.

Note that **%[a-z]** and **%[z-a]** are interpreted as equivalent to **%[abcde...z]**. This is a common **scanf** extension, but note that it is not required by the ANSI specification.

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 is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on **stdin**.

Table R.5 Type Characters for scanf

Character	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to **int**
o	Octal integer	Pointer to **int**
x	Hexadecimal integer[1]	Pointer to **int**
i	Decimal, hexadecimal, or octal integer	Pointer to **int**
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 containing a decimal point, and an optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to **float**
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 format specifier is always interpreted as a hexadecimal number, the input should not include a leading 0x. (If 0x is included, the 0 is interpreted as a hexadecimal input value.)

Return Value

The **scanf** function returns the number of fields that were successfully converted and assigned. The return value may be less than the number requested in the call to **scanf**. The return value does not include fields that were read but not assigned.

The return value is **EOF** if the end-of-file or end-of-string is encountered in the first attempt to read a character.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fscanf, printf, sscanf, vfprintf, vprintf, vsprintf**

Example

```
/* SCANF.C: This program receives formatted input using scanf. */
#include <stdio.h>

void main()
{
   int   i;
   float fp;
   char  c, s[81];
   int   result;

   printf( "Enter an integer, a floating-point number, "
           "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 );
}
```

Output

```
Enter an integer, a floating-point number, a character and a string:
71
98.6
h
White space stops input

The number of fields input is 4
The contents are: 71 98.599998 h White
```

Description

Scrolls the text in a text window.

#include <graph.h>

void _far _scrolltextwindow(short *lines* **);**

lines	Number of lines to scroll

Remarks

The **_scrolltextwindow** function scrolls the text in a text window (previously defined by the **_settextwindow** function). The *lines* argument specifies the number of lines to scroll. A positive value of *lines* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling **_clearscreen(_GWINDOW)**. A value of 0 for *lines* has no effect on the text.

Return Value

None.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_gettextposition, _outmem, _outtext, _settextposition, _settextwindow

Example

```
/* SCRTXWIN.C: This program displays text in text windows and then
 * scrolls, inserts, and deletes lines.
 */

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

void deleteline( void );
void insertline( void );
void status( char *msg );

void main()
{
   short row;
   char  buf[40];

   /* Set up screen for scrolling, and put text window around scroll area. */
   _settextrows( 25 );
   _clearscreen( _GCLEARSCREEN );
```

```
    for( row = 1; row <= 25; row++ )
    {
       _settextposition( row, 1 );
       sprintf( buf, "Line %c                    %2d", row + 'A' - 1, row );
       _outtext( buf );
    }
    getch();
    _settextwindow( 1, 1, 25, 10 );

    /* Delete some lines. */
    _settextposition( 11, 1 );
    for( row = 12; row < 20; row++ )
       deleteline();
    status( "Deleted 8 lines" );

    /* Insert some lines. */
    _settextposition( 5, 1 );
    for( row = 1; row < 6; row++ )
       insertline();
    status( "Inserted 5 lines" );

    /* Scroll up and down. */
    _scrolltextwindow( -7 );
    status( "Scrolled down 7 lines" );
    _scrolltextwindow( 5 );
    status( "Scrolled up 5 lines" );
    _setvideomode( _DEFAULTMODE );
}

/* Delete lines by scrolling them off the top of the current text window.
 * Save and restore original window.
 */
void deleteline()
{
    short left, top, right, bottom;
    struct rccoord rc;

    _gettextwindow( &top, &left, &bottom, &right );
    rc = _gettextposition();
    _settextwindow( rc.row, left, bottom, right );
    _scrolltextwindow( _GSCROLLUP );
    _settextwindow( top, left, bottom, right );
    _settextposition( rc.row, rc.col );
}
```

```
/* Insert some lines by scrolling in blank lines from the top of the
 * current text window. Save and restore original window.
 */
void insertline()
{
   short left, top, right, bottom;
   struct rccoord rc;

   _gettextwindow( &top, &left, &bottom, &right );
   rc = _gettextposition();
   _settextwindow( rc.row, left, bottom, right );
   _scrolltextwindow( _GSCROLLDOWN );
   _settextwindow( top, left, bottom, right );
   _settextposition( rc.row, rc.col );
}

/* Display and clear status in its own window. */
void status( char *msg )
{
   short left, top, right, bottom;
   struct rccoord rc;

   _gettextwindow( &top, &left, &bottom, &right );
   _settextwindow( 1, 50, 2, 80 );
   _outtext( msg );
   getch();
   _clearscreen( _GWINDOW );
   _settextwindow( top, left, bottom, right );
}
```

Description

Searches for a file using environment paths.

#include <stdlib.h>

void _searchenv(char **filename*, char **varname*, char **pathname*);

filename	Name of file to search for
varname	Environment to search
pathname	Buffer to store complete path

Remarks

The **_searchenv** routine searches for the target file in the specified domain. The *varname* variable can be any environment variable which specifies a list of directory paths, such as PATH, LIB, INCLUDE, or other user-defined variables. It is most often PATH, which searches for *filename* on all paths specified in the PATH variable. The **_searchenv** function is case-sensitive, so the *varname* variable should match the case of the environment 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 pointed to by *pathname*. You must ensure that there is sufficient space for the constructed path name. If the *filename* file is not found, *pathname* will contain an empty null-terminated string.

Return Value

The **_searchenv** function does not return a value.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

getenv, putenv

Example

```
/* SEARCHEN.C: This program searches for a file in a directory
 * specified by an environment variable.
 */

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

```
void main()
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";
   /* Search for file in PATH environment variable: */
   _searchenv( searchfile, envvar, pathbuffer );
   if( *pathbuffer != '\0' )
      printf( "Path for %s: %s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

Output

```
Path for CL.EXE: C:\BIN\CL.EXE
```

Description Gets the current values of segment registers.

#include <dos.h>

void segread(struct SREGS ***_segregs_ **);

segregs	Segment-register values

Remarks 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 section 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 None.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **FP_SEG, intdosx, int86x**

Example

```
/* SEGREAD.C: This program gets the current segment values with segread. */

#include <dos.h>
#include <stdio.h>

void main()
{
   struct   SREGS segregs;
   unsigned cs, ds, es, ss;

   /* Read the segment register values */
   segread( &segregs );
   cs = segregs.cs;
   ds = segregs.ds;
   es = segregs.es;
   ss = segregs.ss;
   printf( "CS = 0x%.4x    DS = 0x%.4x    ES = 0x%.4x    SS = 0x%.4x\n",
           cs, ds, es, ss );
}
```

Output

```
CS = 0x0047    DS = 0x0067    ES = 0x0067    SS = 0x0067

CS = 0x2bcc    DS = 0x2ce8    ES = 0x2ba3    SS = 0x2ce8
```

Description

Selects a graphics palette.

#include <graph.h>

short _far _selectpalette(short *number* **);**

number	Palette number

Remarks

The **_selectpalette** function works only under the video modes **_MRES4COLOR** and **_MRESNOCOLOR.** 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.6.

Table R.6 _MRES4COLOR Palette Colors

	Color Index		
Palette Number	**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 gray. 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.7 are available.

Table R.7 _MRESNOCOLOR Mode CGA Palette Colors

	Color Index		
Palette Number	**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.8 are available in the **_MRESNOCOLOR** video mode.

Table R.8 _MRESNOCOLOR Mode EGA Palette Colors

	Color Index		
Palette Number	**Color 1**	**Color 2**	**Color 3**
0	Green	Red	Brown
1	Light green	Light red	Yellow
2	Light cyan	Light red	Yellow

Note that 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.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getvideoconfig**, **_setbkcolor**, **_setvideomode**

Example

```
/* SELPAL.C: This program changes the current CGA palette. */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>

long bkcolor[8] = { _BLACK,  _BLUE,     _GREEN,  _CYAN,
                    _RED,    _MAGENTA,  _BROWN,  _WHITE };
char *bkname [] = { "BLACK", "BLUE",    "GREEN", "CYAN",
                    "RED",   "MAGENTA", "BROWN", "WHITE" };
void main()
{
   int i, j, k;

   if ( !_setvideomode( _MRES4COLOR ) )
   {
      printf( "No palettes available" );
      exit( 1 );
   }
```

```
    for( i = 0; i < 4; i++ )                    /* Palette loop         */
    {
        _selectpalette( i );
        for( k = 0; k < 8; k++ )                /* Background color loop */
        {
            _clearscreen( _GCLEARSCREEN );
            _setbkcolor( bkcolor[k] );
            _settextposition( 1, 1 );
            printf( "Background: %s\tPalette: %d", bkname[k], i );
            for( j = 1; j < 4; j++ )            /* Foreground color loop */
            {
                _setcolor( j );
                _ellipse( _GFILLINTERIOR, 100, j * 30, 220, 80 + (j * 30) );
            }
            getch();
        }
    }
    _setvideomode( _DEFAULTMODE );
}
```

Description

Sets the active page.

#include <graph.h>

short _far _setactivepage(short *page* **);**

page Memory page number

Remarks

For hardware and mode configurations with enough memory to support multiple screen pages, **_setactivepage** specifies the area in memory in which graphics output is written. The *page* argument selects the current active page. The default page number is 0.

Screen animation can be 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 **_gettextcursor**, **_settextcursor**, **_outtext**, **_settextposition**, **_gettextposition**, **_settextcolor**, **_gettextcolor**, **_settextwindow**, and **_wrapon** instead of the standard C-language I/O functions.

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.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getactivepage, **_getvisualpage**, **_setvisualpage**

Example

```
/* PAGE.C illustrates video page functions including:
 *      _getactivepage  _getvisualpage  _setactivepage  _setvisualpage
 */

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
```

```
void main()
{
   short  oldvpage, oldapage, page, row, col, line;
   struct videoconfig vc;
   char   buf[80];

   _getvideoconfig( &vc );
   if( vc.numvideopages < 4 )
       exit( 1 );              /* Fail for OS/2 or monochrome */
   oldapage  = _getactivepage();
   oldvpage  = _getvisualpage();
   _displaycursor( _GCURSOROFF );

   /* Draw arrows in different place on each page. */
   for( page = 1; page < 4; page++ )
   {
      _setactivepage( page );
      _settextposition( 12, 16 * page );
      _outtext( ">>>>>>>>" );
   }

   while( !kbhit() )
      /* Cycle through pages 1 to 3 to show moving image. */
      for( page = 1; page < 4; page++ )
          _setvisualpage( page );
   getch();

   /* Restore original page (normally 0) to restore screen. */
   _setactivepage( oldapage );
   _setvisualpage( oldvpage );
   _displaycursor( _GCURSORON );
}
```

Description Sets the current background color.

#include <graph.h>

long _far _setbkcolor(long *color* **);**

color	Desired color

Remarks The **_setbkcolor** function sets the current background color to the color value *color*.

In a color text mode (such as **_TEXTC80**), **_setbkcolor** accepts (and **_getbkcolor** returns) a color index. The value for the default colors is given in a table in the description of the **_settextcolor** function. For example, **_setbkcolor(2L)** sets the background color to color index 2. The actual color displayed depends on the palette mapping for color index 2. The default is green in a color text mode.

In a color graphics mode (such as **_ERESCOLOR**), **_setbkcolor** accepts (and **_getbkcolor** returns) a color value. The value for the background color is given by the manifest constants defined in the GRAPH.H include file. For example, **_setbkcolor(_GREEN)** sets the background color in a graphics mode to green. These manifest constants are provided as a convenience in defining and manipulating the most common colors. The actual range of colors is, in general, much greater.

In general, whenever an argument is long, it refers to a color value, and whenever it is short, it refers to a color index. The two exceptions are **_setbkcolor** and **_getbkcolor**.

Since the background color is color index 0, the **_remappalette** function will act identically to the **_setbkcolor** function. Unlike **_remappalette**, however, **_setbkcolor** does not require an EGA or VGA environment.

In a text mode, the **_setbkcolor** function does not affect anything already appearing on the display; only the subsequent output is affected. In a graphics mode, it immediately changes all background pixels.

Return Value In text modes, **_setbkcolor** returns the color index of the old background color. In graphics modes, **_setbkcolor** returns the old color value of color index 0. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_getbkcolor, _remappalette, _selectpalette**

Example See the example for **_getcolor.**

Description Controls stream buffering.

#include <stdio.h>

void setbuf(FILE ****stream*, **char ****buffer* **);

stream	Pointer to **FILE** structure
buffer	User-allocated buffer

Remarks The **setbuf** function allows the user to control buffering for *stream*. The *stream* argument must refer to an open file that has not 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, instead of the default system-allocated buffer for the given stream, is used for I/O buffering.

The **stderr** and (in DOS only) **stdaux** streams are unbuffered by default, but may be assigned buffers with **setbuf**.

The **setbuf** function has been subsumed by the **setvbuf** function, which should be the preferred routine for new code. The **setbuf** function is retained for compatibility with existing code.

Return Value None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fclose, fflush, fopen, setvbuf**

Example

```
/* SETBUF.C: 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.
 */

#include <stdio.h>

void main()
{
   char buf[BUFSIZ];
   FILE *stream1, *stream2;
```

```
if( ((stream1 = fopen( "data1", "a" )) != NULL) &&
    ((stream2 = fopen( "data2", "w" )) != NULL) )
{
   /* "stream1" uses user-assigned buffer: */
   setbuf( stream1, buf );
   printf( "stream1 set to user-defined buffer at: %Fp\n", buf );

   /* "stream2" is unbuffered              */
   setbuf( stream2, NULL );
   printf( "stream2 buffering disabled\n" );
   fcloseall();
}
```

Output

```
stream1 set to user-defined buffer at: 0298:0DF2
stream2 buffering disabled
```

Description Sets the clipping region for graphics.

#include <graph.h>

void _far _setcliprgn(short *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

x1, y1	Upper-left corner of clip region
x2, y2	Lower-right corner of clip region

Remarks The **_setcliprgn** function limits the display of subsequent graphics output and font text output to an area of the screen called 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 view coordinate system. Rather, it merely masks the screen.

Note that the **_setcliprgn** function affects graphics and font text output only. To mask the screen for text output, use the **_settextwindow** function.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_settextwindow, _setvieworg, _setviewport, _setwindow**

Example

```
/* SCLIPRGN.C */
#include <stdlib.h>
#include <conio.h>
#include <graph.h>

void main()
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   /* Set clip region, then draw and ellipse larger than the region. */
   _setcliprgn( 0, 0, 200, 125 );
   _ellipse( _GFILLINTERIOR, 80, 50, 240, 190 );
```

```
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Sets the current color.

#include <graph.h>

short _far _setcolor(short *color* **);**

color	Desired color index

Remarks The **_setcolor** function sets the current color index to *color*. The *color* parameter is masked but always within range. The following graphics functions use the current color: **_arc**, **_ellipse**, **_floodfill**, **_lineto**, **_outgtext**, **_pie**, **_rectangle**, and **_setpixel**.

The **_setcolor** function accepts an **int** value as an argument. It is a color index.

The default color index is the highest numbered color index in the current palette.

Note that the **_setcolor** function does not affect the output of the presentation-graphics functions.

Return Value This function returns the previous color. If the function fails (e.g., if used in a text mode), it returns –1.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_arc** functions, **_ellipse** functions, **_floodfill**, **_getcolor**, **_lineto** functions, **_outgtext**, **_pie** functions, **_rectangle** functions, **_selectpalette**, **_setpixel** functions

Example

```
/* GPIXEL.C: This program assigns different colors to randomly
 * selected pixels.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   short xvar, yvar;
   struct videoconfig vc;
```

```
    /* Find a valid graphics mode. */
    if( !_setvideomode( _MAXCOLORMODE ) )
        exit( 1 );
    _getvideoconfig( &vc );

    /* Draw filled ellipse to turn on certain pixels. */
    _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                              vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );

    /* Draw random pixels in random colors... */
    while( !kbhit() )
    {
        /* ...but only if they are already on (inside the ellipse). */
        xvar = rand() % vc.numxpixels;
        yvar = rand() % vc.numypixels;
        if( _getpixel( xvar, yvar ) != 0 )
        {
            _setcolor( rand() % 16 );
            _setpixel( xvar, yvar );
        }
    }

    getch();          /* Throw away the keystroke. */
    _setvideomode( _DEFAULTMODE );
}
```

Description Sets the fill mask.

#include <graph.h>

void _far _setfillmask(unsigned char _far ***mask*** **);**

mask	Mask array

Remarks The **_setfillmask** function sets the current fill mask, which determines the fill pattern. The mask is an 8-by-8 array of bits in which 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 pattern 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 None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_ellipse** functions, **_floodfill**, **_getfillmask**, **_pie** functions, **_rectangle** functions

Example

```
/* GFILLMSK.C: This program illustrates _getfillmask and _setfillmask. */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void ellipsemask( short x1, short y1, short x2, short y2, char _far *newmask );

unsigned char mask1[8] = { 0x43, 0x23, 0x7c, 0xf7, 0x8a, 0x4d, 0x78, 0x39 };
unsigned char mask2[8] = { 0x18, 0xad, 0xc0, 0x79, 0xf6, 0xc4, 0xa8, 0x23 };
char oldmask[8];

void main()
{
   int loop;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
```

```
   /* Set first fill mask and draw rectangle. */
   _setfillmask( mask1 );
   _rectangle( _GFILLINTERIOR, 20, 20, 100, 100 );
   getch();

   /* Call routine that saves and restores mask. */
   ellipsemask( 60, 60, 150, 150, mask2 );
   getch();

   /* Back to original mask. */
   _rectangle( _GFILLINTERIOR, 120, 120, 190, 190 );
   getch();

   _setvideomode( _DEFAULTMODE );
}

/* Draw an ellipse with a specified fill mask. */
void ellipsemask( short x1, short y1, short x2, short y2, char _far *newmask )
{
   unsigned char savemask[8];

   _getfillmask( savemask );                         /* Save mask         */
   _setfillmask( newmask );                          /* Set new mask      */
   _ellipse( _GFILLINTERIOR, x1, y1, x2, y2 );       /* Use new mask      */
   _setfillmask( savemask );                         /* Restore original  */
}
```

Description

Finds a single font.

#include <graph.h>

short _far _setfont(unsigned char _far ***options*** **);**

options	String describing font characteristics

Remarks

The **_setfont** function finds a single font, from the set of registered fonts, that has the characteristics specified by the *options* string. If a font is found, it is made the current font. The current font is used in all subsequent calls to the **_outgtext** function. There can be only one active font at any time.

The *options* string is a set of characters that specifies the desired characteristics of the font. The **_setfont** function searches the list of registered fonts for a font matching the specified characteristics.

The characteristics that may be specified in the *options* string are shown in the list below. Characteristics specified in the *options* string are not case- or position-sensitive.

Characteristic	Description
t'*fontname'*	Typeface.
h*x*	Character height, where *x* is the number of pixels.
w*y*	Character width, where *y* is the number of pixels.
f	Find only a fixed-space font (should not be used with the **p** characteristic).
p	Find only a proportionally spaced font (should not be used with the **f** characteristic).
v	Find only a vector font (should not be used with the **r** characteristic).
r	Find only a raster-mapped (bit-mapped) font (should not be used with the **v** characteristic).
b	Select a best fit font.
n*x*	Select font number *x*, where *x* is less than or equal to the value returned by the **_registerfonts** function. Use this option to "step through" an entire set of fonts.

You can request as many options as desired, except with **n***x*, which should be used alone. If mutually exclusive options are requested (such as the pair **f/p** or **r/v**), the **_setfont** function ignores them. There is no error detection for incompatible parameters used with **n***x*.

Options can be separated by blanks in the *options* string. Any other character is ignored by **_setfont.**

The **t** (the typeface specification) in *options* is specified as a "t" followed by *fontname* in single quotes. Choose *fontname* from the following list:

Fontname	**Description**
Courier	Fixed-width bit-mapped font with serifs
Helv	Sans serif proportional bit-mapped font
Tms Rmn	Proportional bit-mapped font with serifs
Script	Proportional vector-mapped font of slanted characters formed from nearly continuous lines
Modern	Proportional vector-mapped font without serifs
Roman	Proportional vector-mapped font with serifs

A **b** in the *options* field causes the **_setfont** routine to automatically select the "best fit" font that matches the other characteristics you have specified. If the **b** parameter is specified and at least one font is registered, **_setfont** will always be able to set a font and will return 0 to indicate success.

In selecting a font, the **_setfont** routine uses the following precedence (rated from highest precedence to lowest):

1. Pixel height
2. Typeface
3. Pixel width
4. Fixed or proportional font

You can also specify a pixel width and height for fonts. If a nonexistent value is chosen for either, and the **b** option is specified, the **_setfont** function will chose the closest match. A smaller font size has precedence over a larger size. If **_setfont** requests Helv 12 with best fit, and only Helv 10 and Helv 14 are available, **_setfont** will select Helv 10.

If a nonexistent value is chosen for pixel height and width, the **_setfont** function will apply a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if the **r** (raster-mapped font) option is specified, or if a specific typeface is requested and no best fit (**b**) option is specified.

If you specify the **n***x* parameter, **_setfont** will ignore any other specified options and supply only the font number corresponding to *x*.

Note that the font functions affect only the output from the font output function **_outgtext**; no other C run-time output functions are affected by font usage.

Return Value The **_setfont** function returns a 0 to indicate success and a –1 to indicate an error. An error occurs if a request for a specific font fails and the **b** option was not specified, or if fonts have not yet been registered.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getfontinfo**, **_getgtextextent**, **_outgtext**, **_registerfonts**, **_unregisterfonts**

Example See the example for **_outgtext**.

Description

Changes the orientation of font text output.

#include <graph.h>

struct xycoord _far _setgtextvector(short *x*, **short** *y* **);**

x, *y*	Integers specifying font rotation

Remarks

The **_setgtextvector** function sets the current orientation for font text output to the vector specified by *x* and *y*. The current orientation is used in calls to the **_outgtext** function.

The values of *x* and *y* define the vector which determines the direction of rotation of font text on the screen. The text-rotation options are shown below:

(x, y)	Text Orientation
(0, 0)	Unchanged
(1, 0)	Horizontal text (default)
(0, 1)	Rotated 90 degrees counterclockwise
(–1, 0)	Rotated 180 degrees
(0, –1)	Rotated 270 degrees counterclockwise

If other values are input, only the sign of the input is used. For example, (–3, 0) is interpreted as (–1, 0).

Return Value

The **_setgtextvector** function returns the previous vector in a structure of **xycoord** type. If you pass the **_setgtextvector** function the values (0, 0), the function returns the current vector values in the **xycoord** structure.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getfontinfo, **_getgtextextent**, **_grstatus**, **_outgtext**, **_registerfonts**, **_setfont**, **_unregisterfonts**

Example

See the example for **_outgtext**.

Description

Saves the current state of the program.

#include <setjmp.h>

int setjmp(jmp_buf *env* **);**

env	Variable in which environment is stored

Remarks

The **setjmp** function saves a stack environment that can be subsequently restored using **longjmp.** Used together this way, **setjmp** and **longjmp** provide a way to execute a "non-local **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. All variables (except register variables) accessible to the routine receiving control contain the values they had when **setjmp** was called.

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 0, **setjmp** returns 1. There is no error return.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

longjmp

Example

See the example for **_fpreset.**

Description Sets the line style.

#include <graph.h>

void _far _setlinestyle(unsigned short *mask* **);**

mask	Desired line-style mask

Remarks 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* argument 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 0xFFFF (a solid line).

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getlinestyle**, **_lineto** functions, **_rectangle** functions

Example See the example for **_getlinestyle**.

Description Defines the locale.

#include <locale.h>

char *setlocale(int *category*, **const char** **locale* **);**

category	Category affected by locale
locale	Name of the locale that will control the specified category

Remarks The **setlocale** function sets the categories specified by *category* to the locale specified by *locale*. The "locale" refers to the locality (country) for which certain aspects of your program can be customized. Some locale-dependent aspects include the formatting of dates and the display format for monetary values.

The **setlocale** function is used to set or get the program's current entire locale or simply portions of the locale information. The *category* argument specifies which portion of a program's locale information will be used. The manifest constants used for the *category* argument are listed below:

Category	Parts of Program Affected
LC_ALL	All categories listed below.
LC_COLLATE	The **strcoll** and **strxfrm** functions.
LC_CTYPE	The character-handling functions (except for **isdigit** and **isxdigit**, which are unaffected).
LC_MONETARY	Monetary formatting information returned by the **localeconv** function.
LC_NUMERIC	Decimal point character for the formatted output routines (such as **printf**), for the data conversion routines, and for the nonmonetary formatting information returned by the **localeconv** function.
LC_TIME	The **strftime** function.

The *locale* argument is a pointer to a string that specifies the name of the locale. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of "C" specifies the minimal ANSI conforming environment for C translation. This is the only locale supported in Microsoft C, version 6.0.

If the *locale* argument is a null pointer, **setlocale** returns a pointer to the string associated with the category of the program's locale. The program's current locale setting is not changed.

Return Value

If a valid locale and category are given, **_setlocale** returns a pointer to the string associated with the specified category for the new locale. If the locale or category is invalid, the **setlocale** function returns a null pointer and the program's current locale settings are not changed.

The pointer to a string returned by **setlocale** can be used in subsequent calls to restore that part of the program's locale information. Later calls to **setlocale** will overwrite the string.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

localeconv, strcoll, strftime, strxfrm

Description Sets the file translation mode.

#include <fcntl.h>

#include <io.h> Required only for function declarations

int setmode (int *handle*, **int** *mode* **);**

handle	File handle
mode	New translation mode

Remarks The **setmode** function sets to *mode* the translation mode of the file given by *handle*. The mode must be one of the following manifest constants:

Constant	Meaning
O_TEXT	Sets text (translated) mode. Carriage-return–line-feed (CR-LF) combinations are translated into a single line-feed (LF) character on input. Line-feed characters are translated into CR-LF combinations on output.
O_BINARY	Sets 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. If **setmode** is applied to the file handle for a stream, the **setmode** function should be called before any input or output operations are performed on the stream.

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**)

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **creat, fopen, open**

Example

```
/* SETMODE.C: This program uses setmode to change stdin from text
 * mode to binary mode.
 */

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

void main()
{
   int result;

   /* Set "stdin" to have binary mode: */
   result = setmode( fileno( stdin ), O_BINARY );
   if( result == -1 )
      perror( "Cannot set mode" );
   else
      printf( "'stdin' successfully changed to binary mode\n" );
}
```

Output

```
'stdin' successfully changed to binary mode
```

Description Set a pixel to the current color.

#include <graph.h>

short _far _setpixel(short *x*, **short** *y* **);**

short _far _setpixel_w(double *wx*, **double** *wy* **);**

x, *y*	Target pixel
wx, *wy*	Target pixel

Remarks The **_setpixel** and the **_setpixel_w** functions set a pixel at a specified location to the current color.

The **_setpixel** function sets the pixel at the view-coordinate point (*x*, *y*) to the current color.

The **_setpixel_w** function sets the pixel at the window-coordinate point (*wx*, *wy*) 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.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getpixel** functions, **_setcolor**

Example

```
/* GPIXEL.C: This program assigns different colors to randomly
 * selected pixels.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   short xvar, yvar;
   struct videoconfig vc;
```

```
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );

   /* Draw filled ellipse to turn on certain pixels. */
   _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                             vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );

   /* Draw random pixels in random colors... */
   while( !kbhit() )
   {
      /* ...but only if they are already on (inside the ellipse). */
      xvar = rand() % vc.numxpixels;
      yvar = rand() % vc.numypixels;
      if( _getpixel( xvar, yvar ) != 0 )
      {
         _setcolor( rand() % 16 );
         _setpixel( xvar, yvar );
      }
   }

   getch();          /* Throw away the keystroke. */
   _setvideomode( _DEFAULTMODE );
}
```

Description

Sets the current text color.

#include <graph.h>

short _far _settextcolor(short *index* **);**

index Desired color index

Remarks

The **_settextcolor** function sets the current text color to the color index specified by *index*. The default text color is the same as the maximum color index.

The **_settextcolor** routine sets the color for the **_outtext** and **_outmem** functions only. It does not affect the color of the **printf** function or the color of text output with the **_outgtext** font routine. Use the **_setcolor** function to change the color of font output.

In text color mode, you can specify a color index in the range 0–31. The colors in the range 0–15 are interpreted as normal (non-blinking). The normal color range is defined below:

Index	Color	Index	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

Blinking is selected by adding 16 to the normal color value.

In every text mode, including monochrome, **_getvideoconfig** returns the value 32 for the number of available colors. The value 32 indicates the range of values (0–31) accepted by the **_settextcolor** function. This includes sixteen normal colors (0–15) and sixteen blinking colors (16–31). Monochrome text mode has fewer unique display attributes, so some color values are redundant. However, because blinking is selected in the same manner, monochrome text mode has the same range (0–31) as other text modes.

Return Value

The function returns the color index of the previous text color. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_gettextcolor, _outtext**

Example

```
/* OUTTXT.C: This example illustrates text output functions:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition
 */

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

char buffer [80];

void main()
{

   /* Save original foreground, background, and text position */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct rccoord oldpos;

   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );

   /* First time no blink, second time blinking. */
   for( blink = 0; blink <= 16; blink += 16 )
   {
      /* Loop through 8 background colors. */
      for( bgd = 0; bgd < 8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf(buffer, "Back: %d Fore:", bgd );
         _outtext( buffer );
```

```
         /* Loop through 16 foreground colors. */
         for( fgd = 0; fgd < 16; fgd++ )
         {
            _settextcolor( fgd + blink );
            sprintf( buffer, " %2d ", fgd + blink );
            _outtext( buffer );
         }
      }
   }
   getch();

   /* Restore original foreground, background, and text position. */
   _settextcolor( oldfgd );
   _setbkcolor( oldbgd );
   _clearscreen( _GCLEARSCREEN );
   _settextposition( oldpos.row, oldpos.col );
}
```

Description Sets the current cursor attribute.

#include <graph.h>

short _far _settextcursor(short *attr* **);**

attr	Cursor attribute

Remarks The **_settextcursor** function sets the cursor attribute (i.e., the shape) to the value specified by *attr*. The high-order byte of *attr* determines the top line of the cursor within the character cell. The low-order byte of *attr* determines the bottom line of the cursor.

The **_settextcursor** function uses the same format as the BIOS routines in setting the cursor. Typical values for the cursor attribute are listed below:

Attribute	Cursor Shape
0x0707	Underline
0x0007	Full block cursor
0x0607	Double underline
0x2000	No cursor

Note that this function works only in text video modes.

Return Value The function returns the previous cursor attribute, or –1 if an error occurs (such as calling the function in a graphics screen mode).

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **_displaycursor, _gettextcursor**

Example

```
/* DISCURS.C: This program changes the cursor shape using _gettextcursor
 * and _settextcursor, and hides the cursor using _displaycursor.
 */

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

```
void main()
{
   short oldcursor;
   short newcursor = 0x007;        /* Full block cursor */

   /* Save old cursor shape and make sure cursor is on. */
   oldcursor = _gettextcursor();
   _clearscreen( _GCLEARSCREEN );
   _displaycursor( _GCURSORON );
   _outtext( "\nOld cursor shape: " );
   getch();

   /* Change cursor shape. */
   _outtext( "\nNew cursor shape: " );
   _settextcursor( newcursor );
   getch();

   /* Restore original cursor shape. */
   _outtext( "\n" );
   _settextcursor( oldcursor );
}
```

Description

Sets the text position.

#include <graph.h>

struct rccoord _far _settextposition(short *row*, **short** *column* **);**

row, column	New output start position

Remarks

The **_settextposition** function sets the current text position to the display point (*row, column*). The **_outtext** and **_outmem** functions (and standard console I/O routines, such as **printf**) output text at that point.

The **rccoord** structure, defined in GRAPH.H, contains the following elements:

Element	Description
short row	Row coordinate
short col	Column coordinate

Return Value

The function returns the previous text position in an **rccoord** structure, defined in GRAPH.H.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

_gettextposition, _outtext, _settextwindow

Example

```
/* OUTTXT.C: This example illustrates text output functions:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition
 */

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

char buffer [80];

void main()
{
```

```
   /* Save original foreground, background, and text position */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct rccoord oldpos;

   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );

   /* First time no blink, second time blinking. */
   for( blink = 0; blink <= 16; blink += 16 )
   {
      /* Loop through 8 background colors. */
      for( bgd = 0; bgd < 8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf(buffer, "Back: %d Fore:", bgd );
         _outtext( buffer );

         /* Loop through 16 foreground colors. */
         for( fgd = 0; fgd < 16; fgd++ )
         {
            _settextcolor( fgd + blink );
            sprintf( buffer, " %2d ", fgd + blink );
            _outtext( buffer );
         }
      }
   }
   getch();

   /* Restore original foreground, background, and text position. */
   _settextcolor( oldfgd );
   _setbkcolor( oldbgd );
   _clearscreen( _GCLEARSCREEN );
   _settextposition( oldpos.row, oldpos.col );
}
```

Description

Sets the number of screen rows for text modes.

#include <graph.h>

short _far _settextrows(short *rows* **);**

rows	Number of text rows

Remarks

The **_settextrows** function specifies the number of screen rows to be used in text modes.

If the constant **_MAXTEXTROWS** is specified for the *rows* argument, the function will choose the maximum number of rows available. In text modes, this is 50 rows on VGA, 43 on EGA, and 25 on others. In graphics modes that support 30 or 60 rows, **_MAXTEXTROWS** specifies 60 rows.

Return Value

This function returns the numbers of rows set. The function returns 0 if an error occurred.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_getvideoconfig, _setvideomode, _setvideomoderows

Example

```
/* STXTROWS.C: This program attempts to set the screen height. It returns
 * an errorlevel code of 1 (fail) or 0 (success) that could be tested in
 * a batch file.
 */

#include <graph.h>
#include <stdlib.h>

void main( int argc, char **argv )
{
   short rows;

   if( !(rows = atoi( argv[1] )) )
   {
      _outtext( "\nSyntax: STXTROWS [ 25 | 43 | 50 ]\n" );
      exit( 1 );
   }
```

```
   /* Make sure new rows are the same as requested rows. */
   if( _settextrows( rows ) != rows )
   {
      _outtext( "\nInvalid rows\n" );
      exit( 1 );
   }
   else
      exit( 0 );
}
```

Description

Creates a text window.

#include <graph.h>

void _far _settextwindow(short *r1*, **short** *c1*, **short** *r2*, **short** *c2* **);**

r1, *c1*	Upper-left corner of window
r2, *c2*	Lower-right corner of window

Remarks

The **_settextwindow** function specifies a window in row and column coordinates where all text output to the screen is displayed. The arguments (*r1*, *c1*) specify the upper-left corner of the text window, and the arguments (*r2*, *c2*) specify the lower-right corner of the text window.

Text is output from the top of the text window down. When the text window is full, the uppermost line scrolls up out of it.

Note that this function does not affect the output of presentation-graphics text (e.g., labels, axis marks, etc.). It also does not affect the output of the font display routine **_outgtext**. Use the **_setviewport** function to control the display area for presentation graphics or fonts.

Return Value

None.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

_gettextposition, _gettextwindow, _outtext, _settextposition

Example

See the example for **_scrolltextwindow**.

Description

Controls stream buffering and buffer size.

#include <stdio.h>

int setvbuf(FILE **stream*, **char** **buffer*, **int** *mode*, **size_t** *size* **);**

stream	Pointer to **FILE** structure
buffer	User-allocated buffer
mode	Mode of buffering: **_IOFBF** (full buffering), **_IOLBF** (line buffering), **_IONBF** (no buffer)
size	Size of buffer

Remarks

The **setvbuf** function allows the program to control both buffering and buffer size for *stream*. The *stream* must refer to an open file that has not been read from or written to since it was opened. The array pointed to by *buffer* is used as the buffer, unless it is **NULL**, and an automatically allocated buffer *size* bytes long is used.

The mode must be **_IOFBF**, **_IOLBF**, or **_IONBF**. If *mode* is **_IOFBF** or **_IOLBF**, then *size* is used as the size of the buffer. If *mode* is **_IONBF**, the stream is unbuffered and *size* and *buffer* are ignored.

Values for *mode* and their meanings are:

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	Under DOS and OS/2, the same as **_IOFBF**.
_IONBF	No buffer is used, regardless of *buffer* or *size*.

The legal values for *size* are greater than 0 and less than 32,768.

Return Value

The return value for **setvbuf** is 0 if successful, and a nonzero value if an illegal type or buffer size is specified.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fclose, fflush, fopen, setbuf**

Example

```
/* SETVBUF.C: 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.
 */

#include <stdio.h>

void main()
{
   char buf[1024];
   FILE *stream1, *stream2;

   if( ((stream1 = fopen( "data1", "a" )) != NULL) &&
       ((stream2 = fopen( "data2", "w" )) != NULL) )
   {
      if( setvbuf( stream1, buf, _IOFBF, sizeof( buf ) ) != 0 )
         printf( "Incorrect type or size of buffer for stream1\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 buffer for stream2\n" );
      else
         printf( "'stream2' now has no buffer\n" );
      fcloseall();
   }
}
```

Output

```
'stream1' now has a buffer of 1024 bytes
'stream2' now has no buffer
```

Description

Sets the video mode.

#include <graph.h>

short _far _setvideomode(short *mode* **);**

mode	Desired mode

Remarks

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.9 and defined in GRAPH.H.

Table R.9 Manifest Constants for Screen Mode

Mode	Type[1]	Size[2]	Colors[3]	Adapter[4]
_DEFAULTMODE	Hardware default mode			
_MAXRESMODE	Highest resolution in graphics mode			
_MAXCOLORMODE	Maximum colors in graphics mode			
_TEXTBW40	M/T	40 × 25	16	CGA
_TEXTC40	C/T	40 × 25	16	CGA
_TEXTBW80	M/T	80 × 25	16	CGA
_TEXTC80	C/T	80 × 25	6	CGA
_MRES4COLOR	C/G	320 × 200	4	CGA
_MRESNOCOLOR	M/G	320 × 200	4	CGA
_HRESBW	M/G	640 × 200	2	CGA
_TEXTMONO	M/T	80 × 25	1	MDPA
_HERCMONO[5]	Hercules graphics	720 × 348	1	HGC
_MRES16COLOR	C/G	320 × 200	16	EGA
_HRES16COLOR	C/G	640 × 200	16	EGA
_ERESNOCOLOR	M/T	640 × 350	1	EGA
_ERESCOLOR	C/G	640 × 350	16	EGA

Table R.9 *(continued)*

Mode	Type[1]	Size[2]	Colors[3]	Adapter[4]
_VRES2COLOR	C/G	640 × 480	2	VGA
_VRES16COLOR	C/G	640 × 480	16	VGA
_MRES256COLOR	C/G	320 × 200	256	VGA
_ORESCOLOR	C/G	640 × 400	1 of 16	OLIV

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 × rows). For graphics modes, size is given in pixels (horizontal × vertical).
3. For monochrome displays, the number of colors is the number of gray shades.
4. Adapters are the IBM (and compatible) Monochrome Adapter (MDPA), Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA), Hercules-compatible adapter (HGC), and Olivetti-compatible adapter (OLIV).
5. In **_HERCMONO** mode, the text dimensions are 80 columns by 25 rows, with a 9 by 14 character box. The bottom two scan lines of row 25 are not visible.

Note that only standard hardware is described here, but display hardware that is strictly compatible with IBM, Hercules, or Olivetti hardware should also work as described.

_MAXRESMODE and _MAXCOLORMODE

The two special modes **_MAXRESMODE** and **_MAXCOLORMODE** select the highest resolution or greatest number of colors available with the current hardware, respectively. These two modes fail for adapters that do not support graphics modes.

Table R.10 lists the video mode selected for different adapter and monitor combinations when **_MAXRESMODE** or **_MAXCOLORMODE** is specified:

Table R.10 Modes Selected by _MAXRESMODE and _MAXCOLORMODE

Adapter/Monitor	_MAXRESMODE	_MAXCOLORMODE
MDPA	fails	fails
HGC	**_HERCMONO**	**_HERCMONO**
CGA color*	**_HRESBW**	**_MRES4COLOR**
CGA noncolor*	**_HRESBW**	**_MRESNOCOLOR**
OCGA	**_ORESCOLOR**	**_MRES4COLOR**
OEGA color	**_ORESCOLOR**	**_ERESCOLOR**
EGA color 256K	**_HRES16COLOR**	**_HRES16COLOR**
EGA color 64K	**_HRES16COLOR**	**_HRES16COLOR**

Table R.10 *(continued)*

Adapter/Monitor	_MAXRESMODE	_MAXCOLORMODE
EGA ecd 256K	**_ERESCOLOR**	**_ERESCOLOR**
EGA ecd 64K	**_ERESCOLOR**	**_HRES16COLOR**
EGA mono	**_ERESNOCOLOR**	**_ERESNOCOLOR**
MCGA	**_VRES2COLOR**	**_MRES256COLOR**
VGA	**_VRES16COLOR**	**_MRES256COLOR**
OVGA	**_VRES16COLOR**	**_MRES256COLOR**

* Color monitor is assumed if the start-up text mode was TEXTC80 or TEXTC40 or if the start-up mode was graphics mode. Composite or other noncolor CGA monitor is assumed if start-up mode was TEXTBW80 or TEXTBW40.

Hercules Support

You must install the Hercules driver MSHERC.COM before running your program. Type MSHERC to load the driver. This can be done from an AUTOEXEC.BAT file.

If you have both a Hercules monochrome card and a color video card, you should install MSHERC.COM with the /H (/HALF) option. The /H option causes the driver to use one instead of two graphics pages. This prevents the two video cards from attempting to use the same memory. You do not have to use the /H option if you have only a Hercules card. See your Hercules hardware manuals for more details of compatibility.

To use a mouse, you must follow special instructions for Hercules cards in *Microsoft Mouse Programmer's Reference Guide*. (This is sold separately; it is not supplied with either Microsoft C or the mouse package.)

Return Value

The function returns the number of text rows 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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

In OS/2, only text video modes may be selected by **_setvideomode**.

See Also

_getvideoconfig, _settextrows, _setvideomoderows

Example

```
/* SVIDMODE.C: This program sets a video mode from a string given on the
 * command line.
 */
```

```
#include <graph.h>
#include <stdlib.h>
#include <string.h>

short modes[] = { _TEXTBW40,     _TEXTC40,      _TEXTBW80,
                  _TEXTC80,      _MRES4COLOR,   _MRESNOCOLOR,
                  _HRESBW,       _TEXTMONO,     _HERCMONO,
                  _MRES16COLOR,  _HRES16COLOR,  _ERESNOCOLOR,
                  _ERESCOLOR,    _VRES2COLOR,   _VRES16COLOR,
                  _MRES256COLOR, _ORESCOLOR
                };
char *names[] = { "TEXTBW40",     "TEXTC40",      "TEXTBW80",
                  "TEXTC80",      "MRES4COLOR",   "MRESNOCOLOR",
                  "HRESBW",       "TEXTMONO",     "HERCMONO",
                  "MRES16COLOR",  "HRES16COLOR",  "ERESNOCOLOR",
                  "ERESCOLOR",    "VRES2COLOR",   "VRES16COLOR",
                  "MRES256COLOR","ORESCOLOR"
                 };

void error( char *msg );


void main( int argc, char *argv[] )
{
   short i, num = sizeof( modes ) / sizeof( short );
   struct videoconfig vc;

   if( argc < 2 )
      error( "No argument given" );

   /* If matching name found, change to corresponding mode. */
   for( i = 0; i < num; i++ )
   {
      if( !strcmpi( argv[1], names[i] ) )
      {
         _setvideomode( modes[i] );
         _outtext( "New mode is: " );
         _outtext( names[i] );
         exit( 0 );
      }
   }
   error( "Invalid mode string" );
}

void error( char *msg )
{
   _outtext( msg );
   exit( 1 );
}
```

Description Sets the video mode and number of text rows for text modes.

#include <graph.h>

short _far _setvideomoderows(short *mode*, **short** *rows* **);**

mode	Desired mode
rows	Number of text rows

Remarks The **_setvideomoderows** function selects a screen mode for a particular hardware/display combination. The manifest constants for the screen mode are given in the reference pages for **_setvideomode**. The **_setvideomoderows** function also specifies the number of text rows to be used in a text mode. If the constant **_MAXTEXTROWS** is specified for the *rows* argument, the function will choose the maximum number of rows available. In text modes, this is 50 rows on VGA, 43 on EGA, and 25 on others. In graphics modes that support 30 or 60 rows, **_MAXTEXTROWS** specifies 60 rows.

Return Value The **setvideomoderows** function returns the numbers of rows set. The function returns 0 if an error occurred (e.g., if the mode is not supported).

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

In OS/2, only text video modes may be selected by **_setvideomoderows**.

See Also **_getvideoconfig, _settextrows, _setvideomode**

Example

```
/* SVMROWS.C */

#include <stdlib.h>
#include <conio.h>
#include <graph.h>

void main()
{
   struct videoconfig config;
```

```
    /* Set 43-line graphics mode if available. */
    if( !_setvideomoderows( _ERESCOLOR, 43 ) )
    {
       _outtext( "EGA or VGA required" );
       exit( 1 );
    }
    _getvideoconfig( &config );

    /* Set logical origin to center and draw a rectangle. */
    _setlogorg( config.numxpixels / 2 - 1, config.numypixels / 2 - 1 );
    _rectangle( _GBORDER, -80, -50, 80, 50 );

    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Description Moves the view-coordinate origin to the specified physical point.

#include <graph.h>

struct xycoord _far _setvieworg(short *x*, **short** *y* **);**

x, *y*	New origin point

Remarks The **_setvieworg** function moves the view-coordinate origin (0, 0) to the physical point (*x*, *y*). All other view-coordinate points move the same direction and distance.

The **xycoord** structure, defined in GRAPH.H, contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

C 5.1 Difference *This function replaces the **_setlogorg** function.*

Return Value The function returns the physical coordinates of the previous view origin in an **xycoord** structure, defined in GRAPH.H.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getphyscoord, _getviewcoord, _getwindowcoord, _setcliprgn, _setviewport**

Example

```
/* SVORG.C: This program sets the view origin to the center of
 * the screen, then draws a rectangle using the new origin.
 */

#include <stdlib.h>
#include <conio.h>
#include <graph.h>

void main()
{
   struct videoconfig config;
```

```
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &config );

   /* Set view origin to the center of the screen. */
   _setvieworg( config.numxpixels / 2, config.numypixels / 2 );
   _rectangle( _GBORDER, -80, -50, 80, 50 );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Creates a viewport.

#include <graph.h>

void _far _setviewport(short *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

x1, *y1*	Upper-left corner of viewport
x2, *y2*	Lower-right corner of viewport

Remarks The **_setviewport** function redefines the graphics viewport. The **_setviewport** function defines a clipping region in exactly the same manner as **_setcliprgn**, and then sets the view-coordinate 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. Any window transformation done with the **_setwindow** function applies only to the viewport and not to the entire screen.

Return Value None.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_setcliprgn**, **_setvieworg**, **_setwindow**

Example

```
/* SVIEWPRT.C: This program sets a viewport and then draws a rectangle
 * around it and an ellipse in it.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   _setviewport( 100, 100, 200, 200 );
   _rectangle( _GBORDER, 0, 0, 100, 100 );
   _ellipse( _GFILLINTERIOR, 10, 10, 90, 90 );
```

```
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Description Sets the visual page.

#include <graph.h>

short _far _setvisualpage(short *page* **);**

page	Visual page number

Remarks For hardware configurations that have an EGA or a VGA 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 number of the previous visual page. If the function fails, it returns a negative value.

Compatibility ☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also **_getactivepage, _getvisualpage, _setactivepage, _setvideomode**

Example See the example for **_setactivepage.**

Description

Defines a graphics window.

#include <graph.h>

short _far _setwindow(short *finvert*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

finvert	Invert flag
wx1, wy1	Upper-left corner of window
wx2, wy2	Lower-right corner of window

Remarks

The **_setwindow** function defines a window bounded by the specified coordinates. The arguments (*wx1*, *wy1*) specify the upper-left corner of the window, and the arguments (*wx2*, *wy2*) specify the lower-right corner of the window.

The *finvert* argument specifies the direction of the coordinates. If *finvert* is **TRUE**, the *y* axis increases from the screen bottom to the screen top (Cartesian coordinates). If *finvert* is **FALSE**, the *y* axis increases from the screen top to the screen bottom (screen coordinates).

Any window transformation done with the **_setwindow** function applies only to the viewport and not to the entire screen.

If *wx1* equals *wx2* or *wy1* equals *wy2*, the function will fail.

Note that this function does not affect the output of presentation-graphics text (e.g., labels, axis marks, etc.). It also does not affect the output of the font display routine **_outgtext.**

Return Value

The function returns a nonzero value if successful. If the function fails (e.g., if it is not in a graphics mode), it returns 0.

Compatibility

□ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also

_setviewport

Example

```
/* SWINDOW.C: This program illustrates translation between window,
 * view, and physical coordinates. Functions used include:
 *      _setwindow          _getwindowcoord
 *      _getphyscoord       _getviewcoord_wxy
 */
```

```
#include <conio.h>
#include <stdlib.h>
#include <graph.h>

enum boolean { FALSE, TRUE };
enum display { MOVE, DRAW, ERASE };

void main()
{
   struct xycoord view, phys;
   struct _wxycoord oldwin, newwin;
   struct videoconfig vc;
   double xunit, yunit, xinc, yinc;
   short  color, key, fintersect = FALSE, fdisplay = TRUE;

   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );

   /* Set a window using real numbers. */
   _setwindow( FALSE, -125.0, -100.0, 125.0, 100.0 );

   /* Calculate the size of one pixel in window coordinates.
    * Then get the current window coordinates and color.
    */
   oldwin = _getwindowcoord( 1, 1 );
   newwin = _getwindowcoord( 2, 2 );
   xunit = xinc = newwin.wx - oldwin.wx;
   yunit = yinc = newwin.wy - oldwin.wy;
   newwin = oldwin = _getcurrentposition_w();
   color = _getcolor();

   while( 1 )
   {
      /* Set flag according to whether current pixel is on, then
       * turn pixel on.
       */
      if( _getpixel_w( oldwin.wx, oldwin.wy ) == color )
         fintersect = TRUE;
      else
         fintersect = FALSE;
      _setcolor( color );
      _setpixel_w( oldwin.wx, oldwin.wy );
```

```
/* Get and test key. */
key = getch();
switch( key )
{
   case 27:                       /* ESC Quit                  */
      _setvideomode( _DEFAULTMODE );
      exit( 0 );
   case 32:                       /* SPACE    Move no color    */
      fdisplay = MOVE;
      continue;
   case 0:                        /* Extended code - get next */
      key = getch();
      switch( key )
      {
         case 72:                 /* UP          -y            */
            newwin.wy -= yinc;
            break;
         case 77:                 /* RIGHT    +x               */
            newwin.wx += xinc;
            break;
         case 80:                 /* DOWN        +y            */
            newwin.wy += yinc;
            break;
         case 75:                 /* LEFT     -x               */
            newwin.wx -= xinc;
            break;
         case 82:                 /* INS      Draw white       */
            fdisplay = DRAW;
            continue;
         case 83:                 /* DEL      Draw black       */
            fdisplay = ERASE;
            continue;
      }
      break;
}

/* Translate window coordinates to view, view to physical.
 * Then check physical to make sure we're on screen. Update screen
 * and position if we are. Ignore if not.
 */
view = _getviewcoord_wxy( &newwin );
phys = _getphyscoord( view.xcoord, view.ycoord );
if( (phys.xcoord >= 0) && (phys.xcoord < vc.numxpixels) &&
    (phys.ycoord >= 0) && (phys.ycoord < vc.numypixels) )
{
```

```
        /* If display on, draw to new position, else move to new. */
        if( fdisplay != MOVE )
        {
           if( fdisplay == ERASE )
              _setcolor( Ø );
           _lineto_w( newwin.wx, newwin.wy );
        }
        else
        {
           _setcolor( Ø );
           _moveto_w( newwin.wx, newwin.wy );

           /* If there was no intersect, erase old pixel. */
           if( !fintersect )
              _setpixel_w( oldwin.wx, oldwin.wy );
        }
        oldwin = newwin;
     }
     else
        newwin = oldwin;
  }
}
```

Description

Sets the current logical mode for line drawing.

#include <graph.h>

short _far _setwritemode(short *action* **);**

action	Interaction with existing screen image

Remarks

The **_setwritemode** function sets the current logical write mode, which is used when drawing lines with the **_lineto** and **_rectangle** functions.

The *action* argument defines the write mode. The possible values are **_GAND, _GOR, _GPRESET, _GPSET,** and **_GXOR.** See the description of the **_putimage** function for more details on these manifest constants.

Return Value

The **_setwritemode** function returns the previous write mode, or –1 if an error occurs.

Compatibility

☐ ANSI ■ DOS ☐ OS/2 ☐ UNIX ☐ XENIX

See Also

_getwritemode, _grstatus, _lineto functions, **_putimage** functions, **_rectangle** functions, **_setcolor, _setlinestyle**

Example

See the example for **_getwritemode.**

Description

Sets interrupt signal handling.

#include <signal.h>

void (*signal(int *sig*, **void(** **func* **)(int** *sig* **[[, int** *subcode* **]]))) (int** *sig* **);**

sig	Signal value
func	Function to be executed
subcode	Optional subcode to the signal number

Remarks

The **signal** function allows a process to choose one of several ways to handle an interrupt signal from the operating system.

The *sig* argument must be one of the manifest constants described in Table R.11 and defined in SIGNAL.H.

Table R.11 Signals and Responses

Value	Modes	Meaning	Default Action
SIGABRT	Real, protected	Abnormal termination	Terminates the calling program with exit code 3
SIGBREAK	Protected	CTRL+BREAK signal	Terminates the calling program with exit code 3
SIGFPE	Real, protected	Floating-point error	Terminates the calling program with exit code 3
SIGILL	Real, protected	Illegal instruction	Terminates the calling program with exit code 3
SIGINT	Real, protected	CTRL+C signal	Terminates the calling program with exit code 3
SIGSEGV	Real, protected	Illegal storage access	Terminates the calling program with exit code 3
SIGTERM	Real, protected	Termination request	Terminates the calling program with exit code 3
SIGUSR1	Protected	OS/2 process flag A	Signal is ignored
SIGUSR2	Protected	OS/2 process flag B	Signal is ignored
SIGUSR3	Protected	OS/2 process flag C	Signal is ignored

SIGUSR1, **SIGUSR2**, and **SIGUSR3** are user-defined signals which can be sent by means of **DosFlagProcess**. For details, see *Microsoft Operating System/2 Programmer's Reference*.

Note that **SIGILL, SIGSEGV,** and **SIGTERM** are not generated under DOS and **SIGSEGV** is not generated under OS/2. They are included for ANSI compatibility. Thus, you may set signal handlers for these signals via **signal**, and you may also explicitly generate these signals by calling **raise.**

Note also that signal settings are not preserved in child processes created by calls to **exec** or **spawn.** The signal settings are reset to the default in the child process.

The action taken when the interrupt signal is received depends on the value of *func*. The *func* argument must be either a function address or one of the manifest constants defined in SIGNAL.H and listed below:

Value	Meaning
SIG_ACK	Acknowledges receipt of a signal (OS/2 only). This constant is valid only if a user-defined signal handler is installed. Once a process receives a given signal, the operating system does not send any more signals of this type until it receives a **SIG_ACK** acknowledgment from the process. The operating system does not queue up signals of a given type; therefore, if more than one signal of a given type accumulates before the process returns a **SIG_ACK** value, only the most recent signal is sent to the process after the **SIG_ACK** value is received by the operating system. This option has no effect on which handler is installed for a given signal. The manifest constant **SIG_ACK** is not supported for **SIGFPE** signals.
SIG_DFL	Uses system-default response. The system-default response for all signals except **SIGUSR1, SIGUSR2**, and **SIGUSR3** is to abort the calling program. The calling process is terminated with exit code 3, and control returns to DOS or OS/2. If the calling program uses stream I/O, buffers created by the run-time library are not flushed, but buffers created by the operating system are flushed. The default response for **SIGUSR1, SIGUSR2**, and **SIGUSR3** is to ignore the signal.
SIG_ERR	Ignores interrupt signal (OS/2 only). This constant is equivalent to **SIG_IGN,** except that any process that tries to send this signal receives an error. A process may use the **raise** function to send a signal to itself. A different process may send a signal by means of the function **DosFlagProcess** (if the signal is **SIGUSR1, SIGUSR2**, or **SIGUSR3**) or by means of **DosKillProcess** (if the signal is **SIGTERM**).
SIG_IGN	Ignores interrupt signal. This value should never be given for **SIGFPE**, since the floating-point state of the process is left undefined.

Function address	Installs the specified function as the handler for the given signal. For all signals except **SIGFPE** and **SIGUSR***X*, the function is passed the *sig* argument **SIGINT** and executed. For **SIGFPE** signals, the function is passed two arguments; namely **SIGFPE** and the floating-point error code identifying the type of exception that occurred. For **SIGUSR1**, **SIGUSR2**, and **SIGUSR3**, the function is passed two arguments: the signal number and the argument furnished by the **DosFlagProcess** function.

For **SIGFPE**, the function pointed to by *func* is passed two arguments, **SIGFPE** and an integer error subcode, **FPE_***xxx*; then the function is executed. (See the include file FLOAT.H for definitions of the **FPE_***xxx* subcodes.) 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**.) If the function returns, the calling process resumes execution with the floating-point state of the process left undefined.

If the function returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Before the specified function is executed under DOS versions 3.x or earlier, 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 program to reset signals in the called function.

Under OS/2, the signal handler is not reset to the system-default response. Instead, no signals of a given type are received by a process until the process sends a **SIG_ACK** value to the operating system. The program can restore the system-default response from the handler by first sending **SIG_DFL** and then sending **SIG_ACK** to the operating system.

Since signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a C run-time operation is incomplete and in an unknown state. Certain restrictions therefore apply to the C functions that can be used in your signal-handler routine:

1. Do not issue low-level or standard input and output routines (e.g., **printf**, **read**, **write**, and **fread**).
2. Do not call heap routines or any routine that uses the heap routines (e.g., **malloc**, **strdup**, **putenv**).
3. Do not use any C function that generates a system call (e.g., **getcwd**, **time**).

4. Do not use the **longjmp** function unless the interrupt is caused by a floating-point exception (i.e., *sig* is **SIGFPE**). In this case, the program should first reinitialize the floating-point package by means of a call to **_fpreset**.
5. Do not use any overlay routines.

Return Value

The **signal** function returns the previous value of *func* associated with the given signal. For example, if the previous value of *func* was **SIG_IGN**, the return value will be **SIG_IGN**. The one exception to this rule is **SIG_ACK**, which returns the address of the currently installed handler.

A return value of –1 indicates an error, and **errno** is set to **EINVAL**. The possible error causes are an invalid *sig* value, an invalid *func* value (that is, a value that is less than **SIG_ACK** but not defined), or a *func* value of **SIG_ACK** used when no handler is currently installed.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

abort, **exec** functions, **exit**, **_exit**, **_fpreset**, **spawn** functions

Example

```
/* SIGNAL.C illustrates setting up signal interrupt routines. Functions
 * illustrated include signal and raise.
 *
 * Since C I/O functions are not safe inside signal routines, the code
 * uses conditionals to use system-level DOS and OS/2 services. Another
 * option is to set global flags and do any I/O operations outside the
 * signal handler. To compile the OS/2 version, define the symbol OS2.
 */

#include <stdio.h>
#include <conio.h>
#include <signal.h>
#include <process.h>
#include <stdlib.h>
#if defined( OS2 )
   #define INCL_NOCOMMON
   #define INCL_NOPM
   #define INCL_VIO
   #define INCL_KBD
   #include <os2.h>
   #include <string.h>
#else
   #include <dos.h>
   #include <bios.h>
#endif
```

```
void ctrlchandler( void );          /* Prototypes */
void safeout( char *str );
int  safein( void );

void main()
{
   int ch;

   /* Modify CTRL+C behavior. */
   if( signal( SIGINT, ctrlchandler ) == SIG_ERR )
   {
      fprintf( stderr, "Couldn't set SIGINT\n" );
      abort();
   }

   /* Input loop illustrates results. */
   do
   {
      ch = getch();
      if( ch == 0 )
      {
         ch = getch();
         if( ch == 46 )      /* Treat ALT+C like CTRL+C */
            raise( SIGINT );
         else
            printf( "Extended code: %X\n", ch );
      }
      else
         printf( "ASCII code: %X\n", ch );
   } while( ch != 27 );        /* ESC code */
}

/* Handles SIGINT (CTRL+C) interrupt. */
void ctrlchandler()
{
   int c;
   char str[] = " ";

   /* Disallow CTRL+C during handler. */
   signal( SIGINT, SIG_IGN );

   safeout( "User break - abort processing? " );
   c = safein();
   str[0] = c;
   safeout( str );
   safeout( "\r\n" );
   if( (c == 'y') || (c == 'Y') )
      abort();
   else
```

```
      /* The CTRL+C interrupt must be reset to our handler since
       * by default it is reset to the system handler.
       */
      signal( SIGINT, ctrlchandler );
}

/* Outputs a string using system level calls. */
void safeout( char *str )
{
#if defined( OS2 )
   VioWrtTTY( str, strlen( str ), 0 );
#else
   union REGS inregs, outregs;

   inregs.h.ah = 0x0e;
   while( *str )
   {
      inregs.h.al = *str++;
      int86( 0x10, &inregs, &outregs );
   }
#endif
}

/* Inputs a character using system level calls. */
int safein()
{
#if defined( OS2 )
   KBDKEYINFO kki;

   KbdCharIn( &kki, IO_WAIT, 0 );
   return kki.chChar;
#else
   return _bios_keybrd( _KEYBRD_READ ) & 0xff;
#endif
}
```

Output

```
ASCII code: 74
ASCII code: 68
ASCII code: 65
^C
User break - abort processing? n
ASCII code: 62
ASCII code: 1B
```

Description Calculate sines and hyperbolic sines.

#include <math.h>

double sin(double *x* **);**

double sinh(double *x* **);**

long double sinl(long double *x* **);**

long double sinhl(long double *x* **);**

x Angle in radians

Remarks The **sin** and **sinh** functions find the sine and hyperbolic sine of *x*, respectively. The **sinl** and **sinhl** functions are the 80-bit counterparts and use an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value The **sin** functions return 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, the **sin** function 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**.

Compatibility **sin, sinh**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

sinl, sinhl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **acos** functions, **asin** functions, **atan** functions, **cos** functions, **tan** functions

Example

```
/* SINCOS.C: This program displays the sine, hyperbolic sine, cosine,
 * and hyperbolic cosine of pi / 2.
 */
```

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

void main()
{
   double pi = 3.1415926535;
   double x, y;

   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

Output

```
sin( 1.570796 ) = 1.000000
sinh( 1.570796 ) = 2.301299
cos( 1.570796 ) = 0.000000
cosh( 1.570796 ) = 2.509178
```

Description

Opens a file for file sharing.

#include <fcntl.h>

#include <sys\types.h>

#include <sys\stat.h>

#include <share.h>

#include <io.h> Required only for function declarations

int sopen(char **filename*, **int** *oflag*, **int** *shflag* [[, **int** *pmode*]]);

filename	File name
oflag	Type of operations allowed
shflag	Type of sharing allowed
pmode	Permission setting

Remarks

The **sopen** function opens the file specified by *filename* 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 the file FCNTL.H. When two or more constants are used to form the argument *oflag*, the constants are combined 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 *filename* exists.
O_EXCL	Returns an error value if the file specified by *filename* 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.

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 DOS) is not installed, DOS ignores the sharing mode. (See your system documentation for detailed information about sharing modes.)

Constant	Meaning
SH_COMPAT	Sets compatibility mode (not available in OS/2). This is the sharing mode used in the **open** function in DOS.
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. This is the sharing mode used in the **open** function in OS/2.

The **sopen** function should be used only under OS/2 and DOS versions 3.0 and later. Under earlier versions of DOS, the *shflag* argument is ignored.

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 combined 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 DOS and OS/2, 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.

Note that under DOS versions 3.x 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_CREAT | O_RDWR**, *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 solution 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; OS/2 and DOS versions 3.0 and later only).
EEXIST	The **O_CREAT** and **O_EXCL** flags are specified, but the named file already exists.
EINVAL	An invalid *oflag* or *shflag* argument was given.
EMFILE	No more file handles available (too many open files).
ENOENT	File or path name not found.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

close, creat, fopen, _fsopen, open, umask

Example

See the example for **locking**.

Description Create and execute a new child process.

#include <stdio.h>

#include <process.h>

int spawnl(int *mode*, **char** **cmdname*, **char** **arg0*, **char** **arg1*, **... char** **argn*, **NULL);**

int spawnle(int *mode*, **char** **cmdname*, **char** **arg0*, **char** **arg1*, **... char** **argn*, **NULL, char** ***envp* **);**

int spawnlp(int *mode*, **char** **cmdname*, **char** **arg0*, **char** **arg1*, **... char** **argn*, **NULL);**

int spawnlpe(int *mode*, **char** **cmdname*, **char** **arg0*, **char** **arg1*, **... char** **argn*, **NULL, char** ***envp* **);**

int spawnv(int *mode*, **char** **cmdname*, **char** ***argv* **);**

int spawnve(int *mode*, **char** **cmdname*, **char** ***argv*, **char** ***envp* **);**

int spawnvp(int *mode*, **char** **cmdname*, **char** ***argv* **);**

int spawnvpe(int *mode*, **char** **cmdname*, **char** ***argv*, **char** ***envp* **);**

mode	Execution mode for parent process
cmdname	Path name of file to be executed
arg0, ... argn	List of pointers to arguments
argv	Array of pointers to arguments
envp	Array of pointers to environment settings

Remarks The **spawn** family of functions creates and executes a new child process. Enough memory must be available for loading and executing the child process. The *mode* argument determines the action taken by the parent process before and during **spawn.** The following values for *mode* are defined in PROCESS.H:

Value	Meaning
P_DETACH	Continues to execute the parent process; child process is run in the background with no access to the console or keyboard. Calls to **wait** and **cwait** against the child process will fail. This is an asynchronous detached **spawn** and is valid only in OS/2 protected mode.

P_NOWAIT	Continues to execute parent process concurrently with child process (asynchronous **spawn**, valid only in protected mode).
P_NOWAITO	Continues to execute parent process and ignores **wait** and **cwait** calls against child process (asynchronous **spawn**, valid only in protected mode).
P_OVERLAY	Overlays parent process with child, destroying the parent (same effect as **exec** calls).
P_WAIT	Suspends parent process until execution of child process is complete (synchronous **spawn**).

The *cmdname* argument specifies the file which will be executed as the child process, and can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *cmdname* does not have a file-name extension or does not end with a period (.), the **spawn** function first tries the .COM extension, then the .EXE extension, and finally the .BAT extension (or, in OS/2 protected mode, the .CMD extension). This ability to spawn batch files is a new feature in Microsoft C version 6.0.

If *cmdname* has an extension, only that extension is used. If *cmdname* ends with a period, the **spawn** calls search for *cmdname* with no extension. The **spawnlp**, **spawnlpe**, **spawnvp**, and **spawnvpe** routines search for *cmdname* (using the same procedures) in the directories specified by the PATH environment variable.

If *cmdname* contains a drive specifier or any slashes (i.e., if it is a relative path name), the **spawn** call searches only for the specified file and no path searching is done.

Arguments for the Child Process

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 in real mode. The terminating null character (**'\0'**) for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

The argument pointers may be passed as separate arguments (**spawnl**, **spawnle**, **spawnlp**, and **spawnlpe**) or as an array of pointers (**spawnv**, **spawnve**, **spawnvp**, and **spawnvpe**). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is the name of the program as it might be typed on the command line by the user. (A different value will not produce an error.) In real mode, the *argv*[0] value is supplied by the operating system and is the fully qualified path name of the executing program. In protected mode, it is usually the program name as it would be typed on the command line.

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 *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

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 a path name in real mode or to the program name in protected mode, 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.

Environment of the Child Process

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 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.

The **spawn** functions can 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 in real mode (**_C_FILE_INFO** in protected mode).

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 CMD.EXE), this entry remains in the environment. Printing the environment shows graphics characters in the definition string for this entry, since the environment information is passed in binary form in real mode. It should not have any other effect on normal operations. In protected mode, the environment information is passed in text form and therefore contains no graphics characters.

You must explicitly flush (using **fflush** or **flushall**) or close any stream prior to the **spawn** function call.

Starting with Microsoft C version 6.0, you can control whether or not the open file information of a process will be passed to its child processes. The external variable **_fileinfo** (declared in STDLIB.H) controls the passing of **C_FILE_INFO** information. If **_fileinfo** is 0, the **C_FILE_INFO** information is not passed to the child processes. If **_fileinfo** is not 0, **C_FILE_INFO** is passed to child processes.

By default, **_fileinfo** is 0 and thus the **C_FILE_INFO** information is not passed to child processes. There are two ways to modify the default value of **_fileinfo**:

- Link the supplied object file FILEINFO.OBJ into your program. Use the /NOE option to avoid multiple symbol definitions.
- Set the **_fileinfo** variable to a nonzero value directly within your C program.

Return Value

The return value from a synchronous **spawn** (**P_WAIT** specified for *mode*) is the exit status of the child process.

The return value from an asynchronous **spawn** (**P_NOWAIT** or **P_NOWAITO** specified for *mode*) is the process ID. To obtain the exit code for a process spawned with **P_NOWAIT**, you must call the **wait** or **cwait** function and specify the process ID. The exit code cannot be obtained for a process spawned with **P_NOWAITO**.

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 did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an **abort** 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	In DOS, the argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K. In OS/2, the argument list and the space required for environment information combined exceed 32K.
EINVAL	The *mode* argument is invalid.
ENOENT	The file or path name is 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 that 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

The **spawn** functions, with **P_OVERLAY** *mode*, will not work in OS/2 DOS-compatibility mode in programs which are bound with FAPI for dual-mode execution.

Programs linked as DOS mode .EXE files will work, and protected-mode programs will work. The restriction applies only to bound programs in real mode.

In order to ensure proper overlay initialization and termination, do not use the **setjmp** or **longjmp** functions to enter or leave an overlay routine.

See Also **abort, atexit, exec** functions, **exit, _exit, onexit, system**

Example

```
/* SPAWN.C: 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 only has to be in the same path.
 */

#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
};

void main( int argc, char *argv[] )
{
   char *args[4];
   int  result;

   /* Set up parameters to be sent: */
   args[0] = "child";
   args[1] = "spawn??";
   args[2] = "two";
   args[3] = NULL;
   switch (argv[1][0])   /* Based on first letter of argument */
   {
      case '1':
```

```
      spawnl( P_WAIT, argv[2], argv[2], "spawnl", "two", NULL );
      break;
   case '2':
      spawnle( P_WAIT, argv[2], argv[2], "spawnle", "two",
               NULL, my_env );
      break;
   case '3':
      spawnlp( P_WAIT, argv[2], argv[2], "spawnlp", "two", NULL );
      break;
   case '4':
      spawnlpe( P_WAIT, argv[2], argv[2], "spawnlpe", "two",
                NULL, my_env );
      break;
   case '5':
      spawnv( P_OVERLAY, argv[2], args );
      break;
   case '6':
      spawnve( P_OVERLAY, argv[2], args, my_env );
      break;
   case '7':
      spawnvp( P_OVERLAY, argv[2], args );
      break;
   case '8':
      spawnvpe( P_OVERLAY, argv[2], args, my_env );
      break;
   default:
      printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
      exit( 1 );
  }
  printf( "\n\nReturned from SPAWN!\n" );
}
```

Description

Breaks a path name into components.

#include <stdlib.h>

void _splitpath(char ***path*****, char** ***drive*****, char** ***dir*****, char** ***fname*****, char** ***ext*** **);**

path	Full path name
drive	Drive letter
dir	Directory path
fname	File name
ext	File extension

Remarks

The **_splitpath** routine breaks a full path name into its 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 manifest constants **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_FNAME**, and **_MAX_EXT**, defined in STDLIB.H. The other arguments point to the 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*.

Return Value

None.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also _fullpath, _makepath

Example

```
/* MAKEPATH.C */
#include <stdlib.h>
#include <stdio.h>

void main()
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath( path_buffer, "c", "\\c60\\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( "  Filename: %s\n", fname );
   printf( "  Ext: %s\n", ext );
}
```

Output

```
Path created with _makepath: c:\c60\clibref\makepath.c

Path extracted with _splitpath:
  Drive: c:
  Dir: \c60\clibref\
  Filename: makepath
  Ext: .c
```

Description

Writes formatted data to a string.

#include <stdio.h>

int sprintf(char ***buffer*, **const char** ***format* [[, *argument*]] ... **);**

buffer	Storage location for output
format	Format-control string
argument	Optional arguments

Remarks

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 **printf** for a description of the format and arguments.) A null character 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 character.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

fprintf, printf, sscanf

Example

```
/* SPRINTF.C: This program uses sprintf to format various data and
 * place them in the string named buffer.
 */

#include <stdio.h>

void main()
{
   char  buffer[200], s[] = "computer", c = 'l';
   int   i = 35, j;
   float fp = 1.7320534;
```

```
   /* Format and print various data: */
   j  = sprintf( buffer,     "\tString:    %s\n", s );
   j += sprintf( buffer + j, "\tCharacter: %c\n", c );
   j += sprintf( buffer + j, "\tInteger:   %d\n", i );
   j += sprintf( buffer + j, "\tReal:      %f\n", fp );

   printf( "Output:\n%s\ncharacter count = %d\n", buffer, j );
}
```

Output

```
Output:
        String:    computer
        Character: l
        Integer:   35
        Real:      1.732053

character count = 71
```

Description Calculates the square root.

#include <math.h>

double sqrt(double *x*);

long double sqrtl(long double *x*);

x	Nonnegative floating-point value

Remarks The **sqrt** functions calculate the square root of *x*. The **sqrtl** function is the 80-bit counterpart and uses an 80-bit, 10-byte coprocessor form of arguments and return values.

Return Value The **sqrt** functions return 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** or **_matherrl** routine.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **exp, log, matherr, pow**

Example

```
/* SQRT.C: This program calculates a square root. */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
   double question = 45.35, answer;

   answer = sqrt( question );
   if( errno == EDOM )
      printf( "Domain error\n" );
   else
      printf( "The square root of %.2f is %.2f\n", question, answer );
}
```

Output

```
The square root of 45.35 is 6.73
```

Description Sets a random starting point.

#include <stdlib.h>	Required only for function declarations

void srand(unsigned int *seed* **);**

seed	Seed for random-number generation

Remarks 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 that are generated. Calling **rand** before any call to **srand** will generate the same sequence as calling **srand** with *seed* passed as 1.

Return Value None.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **rand**

Example

```
/* RAND.C: This program seeds the random number generator with the
 * time, then displays 20 random integers.
 */

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

void main()
{
   int i;

   /* Seed the random number generator with current time so that
    * the numbers will be different every time we run.
    */
   srand( (unsigned)time( NULL ) );
```

```
   /* Display 10 numbers. */
   for( i = 0; i < 10; i++ )
       printf( "  %6d\n", rand() );
}
```

Output

```
   19471
   16395
    8268
   15582
    6489
   28356
   27042
    5276
   23070
   10930
```

Description Reads formatted data from a string.

#include <stdio.h>

int sscanf(const char ***buffer*****, const char** ***format*** **[[, *argument*]] ...);**

buffer	Stored data
format	Format-control string
argument	Optional arguments

Remarks 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 **scanf** 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fscanf, scanf, sprintf**

Example

```
/* SSCANF.C: This program uses sscanf to read data items from
 * a string named tokenstring, then displays them.
 */

#include <stdio.h>

void main()
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;
```

```
   /* 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 );
   printf( "Character = %c\n", c );
   printf( "Integer:  = %d\n", i );
   printf( "Real:     = %f\n", fp );
}
```

Output

```
String    = 15
Character = 1
Integer:  = 15
Real:     = 15.000000
```

Description Gets the size of the stack available.

#include <malloc.h> Required only for function declarations

size_t stackavail(void);

Remarks 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

Example

```
/* ALLOCA.C: This program checks the stack space available before
 * and after using the alloca function to allocate space on the stack.
 */

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

void main()
{
   char *buffer;

   printf( "Bytes available on stack: %u\n", stackavail() );

   /* Allocate memory for string. */
   buffer = alloca( 120 * sizeof( char ) );
   printf( "Enter a string: " );
   gets( buffer );
   printf( "You entered: %s\n", buffer );

   printf( "Bytes available on stack: %u\n", stackavail() );
}
```

Output

```
Bytes available on stack: 2028
Enter a string: How much stack space will this string take?
You entered: How much stack space will this string take?
Bytes available on stack: 1902
```

Description

Gets status information on a file.

#include <sys\stat.h>

#include <sys\types.h>

int stat(char ***pathname*****, struct stat** ***buffer* **);**

pathname	Path name of existing file
buffer	Pointer to structure that receives results

Remarks

The **stat** function obtains information about the file or directory specified by *pathname* and stores it in the structure pointed to by *buffer*. The **stat** structure, defined in the file SYS\STAT.H, includes 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	Drive number of the disk containing the file (same as **st_rdev**). Real mode only.
st_mode	Bit mask for file-mode information. The **S_IFDIR** bit is set if *pathname* specifies a directory; the **S_IFREG** bit is set if *pathname* specifies an ordinary file. User read/write bits are set according to the file's permission mode; user execute bits are set according to the file-name extension.
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**). Real mode only.
st_size	Size of the file in bytes.

Note that if *pathname* refers to a device, the size and time fields in the **stat** structure are not meaningful.

Return Value The **stat** function returns the value 0 if the file-status information is obtained. A return value of –1 indicates an error; also, **errno** is set to **ENOENT**, indicating that the file name or path name could not be found.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **access, fstat**

Example

```
/* STAT.C: This program uses the stat function to report information
 * about the file named STAT.C.
 */

#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>

void main()
{
   struct stat buf;
   int  fh, result;
   char buffer[] = "A line to output";

   /* Get data associated with "stat.c": */
   result = stat( "stat.c", &buf );

   /* Check if statistics are valid: */
   if( result != 0 )
      perror( "Problem getting information" );
   else
   {
      /* Output some of the statistics: */
      printf( "File size     : %ld\n", buf.st_size );
      printf( "Drive         : %c:\n", buf.st_dev + 'A' );
      printf( "Time modified : %s", ctime( &buf.st_atime ) );
   }
}
```

Output

```
File size     : 761
Drive         : C:
Time modified : Wed Jun 14 12:20:08 1989
```

Description

Gets the floating-point status word.

#include <float.h>

unsigned int _status87(void);

Remarks

The **_status87** function gets the floating-point status word. The status word is a combination of the 8087/80287/80387 status word and other conditions detected by the 8087/80287/80387 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 that 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.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

_clear87, **_control87**

Example

```
/* STATUS87.C: This program creates various floating-point errors and
 * then uses _status87 to display messages indicating these problems.
 */

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

void main()
{
   double a = 1e-40, b;
   float  x, y;

   printf( "Status = %.4x - clear\n",_status87() );

   /* Assignment 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();
}
```

Output

```
Status = 0000 - clear
Status = 0030 - inexact, underflow
Status = 0032 - inexact underflow, denormal
```

Description

Append a string.

#include <string.h> Required only for function declarations

char *strcat(char ****string1*, **const char** ****string2* **);**

char _far * _far _fstrcat(char _far ****string1*, **const char _far** ****string2* **);**

string1	Destination string
string2	Source string

Remarks

The **strcat** and **_fstrcat** functions append *string2* to *string1*, terminate the resulting string with a null character, and return a pointer to the concatenated string (*string1*).

The **strcat** and **_fstrcat** 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 **_fstrcat** function is a model-independent (large-model) form of the **strcat** function. The behavior and return value of **_fstrcat** are identical to those of the model-dependent function **strcat**, with the exception that the arguments and return values are far pointers.

Return Value

The return values for these functions are described above.

Compatibility

strcat

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrcat

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strncat, strncmp, strncpy, strnicmp, strrchr, strspn

Example

```
/* STRCPY.C: This program uses strcpy and strcat to build a phrase. */

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

```
void main()
{
   char string[80];

   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

Output

```
String = Hello world from strcpy and strcat!
```

Description Find a character in a string.

#include <string.h>	Required only for function declarations

char *strchr(const char *string, int c);

char _far * _far _fstrchr(const char _far *string, int c);

string	Source string
c	Character to be located

Remarks The **strchr** and **_fstrchr** functions return 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 **strchr** and **_fstrchr** 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.

The **_fstrchr** function is a model-independent (large-model) form of the **strchr** function. The behavior and return value of **_fstrchr** are identical to those of the model-dependent function **strchr**, with the exception that the arguments and return values are far.

Return Value The return values for these functions are described above.

Compatibility **strchr**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrchr

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strcspn, strncat, strncmp, strncpy, strnicmp, strpbrk, strrchr, strspn, strstr**

Example

```
/* STRCHR.C: This program illustrates searching for a character with
 * strchr (search forward) or strrchr (search backward).
 */
#include <string.h>
#include <stdio.h>
```

```
int  ch = 'r';
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

void main()
{
   char *pdest;
   int result;

   printf( "String to be searched: \n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );
   printf( "Search char:\t%c\n", ch );

   /* Search forward. */
   pdest = strchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );

   /* Search backward. */
   pdest = strrchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tlast %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );

}
```

Output

```
String to be searched:
                The quick brown dog jumps over the lazy fox
                         1         2         3         4         5
                12345678901234567890123456789012345678901234567890

Search char:    r
Result: first r found at position 12

Result: last r found at position 30
```

Description

Compare strings.

#include <string.h>	Required only for function declarations

int strcmp(const char **string1*, **const char** **string2* **);**

int _far _fstrcmp(const char _far **string1*, **const char _far** **string2* **);**

string1	String to compare
string2	String to compare

Remarks

The **strcmp** and **_fstrcmp** functions compare *string1* and *string2* lexicographically and return 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 **strcmp** and **_fstrcmp** 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.

The **_fstrcmp** function is a model-independent (large-model) form of the **strcmp** function. The behavior and return value of **_fstrcmp** are identical to those of the model-dependent function **strcmp**, with the exception that the arguments are far pointers.

The **strcmpi** and **stricmp** functions are case-insensitive versions of **strcmp**.

Return Value

The return values for these functions are described above.

Compatibility

strcmp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrcmp

□ ANSI ■ DOS ■ OS/2 □ UNIX ■ XENIX

See Also memcmp, memicmp, strncat, strncmp, strncpy, strnicmp, strrchr, strspn

Example

```
/* STRCMP.C */
#include <string.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";

void main()
{
   char tmp[20];
   int result;

   /* Case sensitive */
   printf( "Compare strings:\n\t%s\n\t%s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "\tstrcmp:   String 1 is %s string 2\n", tmp );

   /* Case insensitive (could use equivalent stricmp) */
   result = strcmpi( string1, string2 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "\tstrcmpi:  String 1 is %s string 2\n", tmp );
}
```

Output

```
Compare strings:
        The quick brown dog jumps over the lazy fox
        The QUICK brown dog jumps over the lazy fox

        strcmp:   String 1 is greater than string 2
        strcmpi:  String 1 is equal to string 2
```

Description

Compares strings using locale-specific information.

#include <string.h>	Required only for function declarations

int strcoll(const char ****string1*, const char ****string2* **);**

string1	String to compare
string2	String to compare

Remarks

The **strcoll** 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 **strcoll** function operates on null-terminated strings. The string arguments to these functions are expected to contain a null character (**'\0'**) marking the end of the string.

The **strcoll** function differs from **strcmp** in that it uses locale-specific information to provide locale-specific collating sequences.

Return Value

The return value for this function is described above.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

localeconv, setlocale, strcmp, strncmp, strxfrm

Description Copy a string.

#include <string.h> Required only for function declarations

char *strcpy(char ***_string1_, const char ***_string2_ **);**

char _far * _far _fstrcpy(char _far ***_string1_, const char _far ***_string2_ **);**

string1	Destination string
string2	Source string

Remarks The **strcpy** function copies _string2_, including the terminating null character, to the location specified by _string1_, and returns _string1_.

The **strcpy** and **_fstrcpy** 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 **_fstrcpy** function is a model-independent (large-model) form of the **strcpy** function. The behavior and return value of **_fstrcpy** are identical to those of the model-dependent function **strcpy**, with the exception that the arguments and return values are far pointers.

Return Value The return values for these functions are described above.

Compatibility **strcpy**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrcpy

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strcat, strcmp, strncat, strncmp, strncpy, strnicmp, strrchr, strspn**

Example

```
/* STRCPY.C: This program uses strcpy and strcat to build a phrase. */

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

```
void main()
{
   char string[80];

   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

Output

```
String = Hello world from strcpy and strcat!
```

Description

Find a substring in a string.

#include <string.h> Required only for function declarations

size_t strcspn(const char ****string1*, **const char** ****string2* **);**

size_t _far _fstrcspn(const char _far ****string1*, **const char _far** ****string2* **);**

string1	Source string
string2	Character set

Remarks

The **strcspn** functions return the index of the first character in *string1* belonging to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* consisting 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 **strcspn** and **_fstrcspn** 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.

The **_fstrcspn** function is a model-independent (large-model) form of the **strcspn** function. The behavior and return value of **_fstrcspn** are identical to those of the model-dependent function **strcspn**, with the exception that the arguments and return values are far.

Return Value

The return values for these functions are described above.

Compatibility

strcspn

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrcspn

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strncat, strncmp, strncpy, strnicmp, strrchr, strspn

Example

```
/* STRCSPN.C */
#include <string.h>
#include <stdio.h>
```

```
void main()
{
   char string[] = "xyzabc";
   int  pos;

   pos = strcspn( string, "abc" );
   printf( "First a, b or c in %s is at character %d\n", string, pos );
}
```

Output

```
First a, b or c in xyzabc is at character 3
```

Description Copies a date to a buffer.

#include <time.h>

char *_strdate(char *_datestr_);

datestr	Current date

Remarks The **_strdate** function copies the date to the buffer pointed to by *datestr*, 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 The **_strdate** function returns a pointer to the resulting text string *datestr*.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **asctime, ctime, gmtime, localtime, mktime, time, tzset**

Example

```
/* STRTIME.C */
#include <time.h>
#include <stdio.h>

void main()
{
   char dbuffer [9];
   char tbuffer [9];

   _strdate( dbuffer );
   printf( "The current date is %s \n", dbuffer );
   _strtime( tbuffer );
   printf( "The current time is %s \n", tbuffer );

}
```

Output

```
The current date is 06/20/89
The current time is 09:33:13
```

Description Duplicate strings.

#include <string.h> Required only for function declarations

char *strdup(const char ***_string_ **);

char _far * _far _fstrdup(const char _far ***_string_ **);

char _near * _far _nstrdup(const char _far ***_string_ **);

string Source string

Remarks 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.

The **_fstrdup** and **_nstrdup** functions provide complete control over the heap used for string duplication. The **strdup** function returns a pointer to a copy of the string argument. The space for the string is allocated from the heap specified by the memory model in use. In large-data models (that is, compact-, large-, and huge-model programs), **strdup** allocates space from the far heap. In small-data models (tiny-, small-, and medium-model programs), **strdup** allocates space from the near heap.

The **strdup, _fstrdup,** and **_nstrdup** 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.

The **_fstrdup** function returns a far pointer to a copy of the string allocated in far memory (the far heap). As with the other model-independent functions, the syntax and semantics of these functions correspond to those of **strdup** except for the sizes of the arguments and return values. The **_nstrdup** function returns a near pointer to a copy of the string allocated in the near heap (in the default data segment).

Return Value The return values for these functions are described above.

Compatibility **strdup, _fstrdup, _nstrdup**

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **strcat, strcmp, strncat, strncmp, strncpy, strnicmp, strrchr, strspn**

Example

```
/* STRDUP.C */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

void main()
{
   char buffer[] = "This is the buffer text";
   char *newstring;

   printf( "Original: %s\n", buffer );
   newstring = strdup( buffer );
   printf( "Copy:     %s\n", newstring );
}
```

Output

```
Original: This is the buffer text
Copy:     This is the buffer text
```

Description

Gets a system error message (**strerror**) or prints a user-supplied error message (**_strerror**).

#include <string.h>	Required only for function declarations

char *strerror(int *errnum* **);**

char *_strerror(char **string* **);**

errnum	Error number
string	User-supplied message

Remarks

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 newline 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 newline 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 an **fprintf** statement, as shown in the following lines:

```
if ((access("datafile",2)) == -1)
     fprintf(_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 the variable **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 that 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; it 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.

Compatibility

strerror

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

_strerror

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

clearerr, ferror, perror

Example

See the example for **perror**.

Description

Formats a time string.

#include <time.h> Required only for function declarations

size_t strftime(char **string*, **size_t** *maxsize*, **const char** **format*,
const struct tm **timeptr* **);**

string	Output string
maxsize	Maximum length of string
format	Format control string
timeptr	**tm** data structure

Remarks

The **strftime** function formats the **tm** time value in *timeptr* according to the supplied *format* argument and stores the result in the buffer *string*. At most, *maxsize* characters are placed in the string.

The *format* argument consists of one or more codes; as in **printf**, the formatting codes are preceded by a **%** sign. Characters that do not begin with a **%** sign are copied unchanged to *string*. The **LC_TIME** category of the current locale affects the output formatting of **strftime.**

The formatting codes for **strftime** are listed below:

Format	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for the locale
%d	Day of the month as a decimal number (01 – 31)
%H	Hour in 24-hour format (00 – 23)
%I	Hour in 12-hour format (01 – 12)
%j	Day of the year as a decimal number (001 – 366)
%m	Month as a decimal number (01 – 12)
%M	Minute as a decimal number (00 – 59)

%p	Current locale's AM/PM indicator for a 12-hour clock
%S	Second as a decimal number (00 – 61)
%U	Week of the year as a decimal number; Sunday is taken as the first day of the week (00 – 53)
%w	Weekday as a decimal number (0 – 6; Sunday is 0)
%W	Week of the year as a decimal number; Monday is taken as the first day of the week (00 – 53)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without the century as a decimal number (00 – 99)
%Y	Year with the century as a decimal number
%z	Time zone name or abbreviation; no characters if time zone is unknown
%%	Percent sign

Return Value

The **strftime** function returns the number of characters placed in *string* if the total number of resulting characters, including the terminating null, is not more than *maxsize*.

Otherwise, **strftime** returns 0, and the contents of the string are indeterminate.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

localeconv, setlocale, strxfrm

Example

See the example for **perror**.

Description Compare strings without regard to case.

#include <string.h>	Required only for function declarations

int stricmp(const char ****string1*, **const char ****string2* **);

int _far _fstricmp(const char _far ****string1*, **const char _far ****string2* **);

string1	String to compare
string2	String to compare

Remarks The **stricmp** and **_fstricmp** functions compare *string1* and *string2* lexicographically and return 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 **stricmp** and **_fstricmp** 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.

The **_fstricmp** function is a model-independent (large-model) form of the **stricmp** function. The behavior and return value of **_fstricmp** are identical to those of the model-dependent function **stricmp**, with the exception that the arguments are far pointers.

The **strcmp** function is a case-sensitive version of **stricmp**.

Return Value The return values for these functions are described above.

Compatibility **stricmp**

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstricmp

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **memcmp, memicmp, strcat, strcpy, strncat, strncmp, strncpy, strnicmp, strrchr, strset, strspn**

Example See the example for **strcmp**.

Description

Get the length of a string.

#include <string.h> Required only for function declarations

size_t strlen(const char ****string*** **);**

size_t _fstrlen(const char _far ****string*** **);**

string Null-terminated string

Remarks

The **strlen** and **_fstrlen** functions return the length in bytes of *string*, not including the terminating null character (**'\0'**).

The **_fstrlen** function is a model-independent (large-model) form of the **strlen** function. The behavior and return value of **_fstrlen** are identical to those of the model-dependent function **strlen**, with the exception that the argument is a far pointer.

Return Value

These functions return the string length. There is no error return.

Compatibility

strlen

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrlen

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

Example

```
/* STRLEN.C */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
```

```
void main()
{
   char buffer[61] = "How long am I?";
   int  len;

   len = strlen( buffer );
   printf( "'%s' is %d characters long\n", buffer, len );
}
```

Output

```
'How long am I?' is 14 characters long
```

Description

Convert a string to lowercase.

#include <string.h> Required only for function declarations

char *strlwr(char ***string* **);

char _far * _far _fstrlwr(char _far ***string* **);

string	String to be converted

Remarks

The **strlwr** and **_fstrlwr** functions convert any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

The **_fstrlwr** function is a model-independent (large-model) form of the **strlwr** function. The behavior and return value of **_fstrlwr** are identical to those of the model-dependent function **strlwr**, with the exception that the argument and return values are far pointers.

Return Value

These functions return a pointer to the converted string. There is no error return.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strupr

Example

```
/* STRLWR.C: This program uses strlwr and strupr to create
 * uppercase and lowercase copies of a mixed-case string.
 */

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

void main()
{
   char string[100] = "The String to End All Strings!";
   char *copy1, *copy2;

   copy1 = strlwr( strdup( string ) );
   copy2 = strupr( strdup( string ) );
   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
}
```

Output

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

Description Appends characters of a string.

#include <string.h>	Required only for function declarations

char *strncat(char ***string1*****, const char** ***string2*****, size_t** *count* **);**

char _far * _far _fstrncat(char _far ***string1*****, const char _far** ***string2*****, size_t** *count* **);**

string1	Destination string
string2	Source string
count	Number of characters appended

Remarks The **strncat** and **_fstrncat** functions append, at most, the first *count* characters of *string2* to *string1*, terminate the resulting string with a null character ('**\0**'), and return a pointer to the concatenated string (*string1*). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

The **_fstrncat** function is a model-independent (large-model) form of the **strncat** function. The behavior and return value of **_fstrncat** are identical to those of the model-dependent function **strncat**, with the exception that all the pointer arguments and return values are far pointers.

Return Value The return values for these functions are described above.

Compatibility **strncat**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrncat

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strcat, strcmp, strcpy, strncmp, strncpy, strnicmp, strrchr, strset, strspn**

Example

```
/* STRNCAT.C */
#include <string.h>
#include <stdio.h>
```

```
void main()
{
   char string[80] = "This is the initial string!";
   char suffix[] = " extra text to add to the string...";

   /* Combine strings with no more than 19 characters of suffix:  */
   printf( "Before: %s\n", string );
   strncat( string, suffix, 19 );
   printf( "After:  %s\n", string );

}
```

Output

```
Before: This is the initial string!
After:  This is the initial string! extra text to add
```

Description

Compare characters of two strings.

#include <string.h> Required only for function declarations

int strncmp(const char **string1*, **const char** **string2*, **size_t** *count* **);**

int _far _fstrncmp(const char _far **string1*, **const char _far** **string2*, **size_t** *count* **);**

string1	String to compare
string2	String to compare
count	Number of characters compared

Remarks

The **strncmp** and **_fstrncmp** functions lexicographically compare, at most, the first *count* characters of *string1* and *string2* and return a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* equivalent to *string2*
> 0	*string1* greater than *string2*

The **strnicmp** function is a case-insensitive version of **strncmp**.

The **_fstrncmp** function is a model-independent (large-model) form of the **strncmp** function. The behavior and return value of **_fstrncmp** are identical to those of the model-dependent function **strncmp**, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

Compatibility

strncmp

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrncmp

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strcat, strcmp, strcpy, strncat, strncpy, strrchr, strset, strspn

Example

```
/* STRNCMP.C */
#include <string.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

void main()
{
   char tmp[20];
   int result;

   printf( "Compare strings:\n\t\t%s\n\t\t%s\n\n", string1, string2 );

   printf( "Function:\tstrncmp (first 10 characters only)\n" );
   result = strncmp( string1, string2 , 10 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

   printf( "Function:\tstrnicmp (first 10 characters only)\n" );
   result = strnicmp( string1, string2, 10 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

}
```

Output

```
Compare strings:
                The quick brown dog jumps over the lazy fox
                The QUICK brown fox jumps over the lazy dog

Function:       strncmp (first 10 characters only)
Result:         String 1 is greater than string 2

Function:       strnicmp (first 10 characters only)
Result:         String 1 is equal to string 2
```

Description Copy characters of one string to another.

#include <string.h> Required only for function declarations

char *strncpy(char ***_string1_, const char ***_string2_**, size_t** _count_ **);**

char _far * _far _fstrncpy(char _far ***_string1_, const char _far ***_string2_**, size_t** _count_ **);**

string1	Destination string
string2	Source string
count	Number of characters copied

Remarks The **strncpy** and **_fstrncpy** functions copy *count* characters of *string2* to *string1* and return *string1*. If *count* is less than the length of *string2*, a null character (**'\0'**) is not appended automatically to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (**'\0'**) up to length *count*.

Note that the behavior of **strncpy** and **_fstrncpy** is undefined if the address ranges of the source and destination strings overlap.

The **_fstrncpy** function is a model-independent (large-model) form of the **strncpy** function. The behavior and return value of **_fstrncpy** are identical to those of the model-dependent function **strncpy**, with the exception that all the arguments and return values are far.

Return Value The return values for these functions are described above.

Compatibility **strncpy**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrncpy

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strcat, strcmp, strcpy, strncat, strncmp, strnicmp, strrchr, strset, strspn**

Example

```
/* STRNCPY.C */
#include <string.h>
#include <stdio.h>

void main()
{
   char string[100] = "Cats are nice usually";

   printf("Before: %s\n", string );
   strncpy( string, "Dogs", 4 );
   strncpy( string + 9, "mean", 4 );
   printf("After:  %s\n", string );
}
```

Output

```
Before: Cats are nice usually
After:  Dogs are mean usually
```

Description

Compare characters of two strings without regard to case.

#include <string.h> Required only for function declarations

int strnicmp(const char ***string1*, **const char ***string2*, **size_t *count* **);**

**int _far _fstrnicmp(const char _far ***string1*, **const char _far ***string2*,
size_t *count* **);**

string1	String to compare
string2	String to compare
count	Number of characters compared

Remarks

The **strnicmp** and **_fstrnicmp** functions lexicographically compare (without regard to case), at most, the first *count* characters of *string1* and *string2* and return a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* equivalent to *string2*
> 0	*string1* greater than *string2*

The **strncmp** function is a case-sensitive version of **strnicmp.**

The **_fstrnicmp** function is a model-independent (large-model) form of the **strnicmp** function. The behavior and return value of **_fstrnicmp** are identical to those of the model-dependent function **strnicmp**, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

Compatibility

☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also

strcat, strcmp, strcpy, strncat, strncpy, strrchr, strset, strspn

Example

See the example for **strncmp.**

Description

Initialize characters of a string to a given character.

#include <string.h> Required only for function declarations

char *strnset(char ***_string_, int** _c_**, size_t** _count_ **);**

char _far * _far _fstrnset(char _far ***_string_, int** _c_**, size_t** _count_ **);**

string	String to be initialized
_c_	Character setting
count	Number of characters set

Remarks

The **strnset** and **_fstrnset** functions set, at most, the first _count_ characters of _string_ to the character _c_ and return a pointer to the altered string. If _count_ is greater than the length of _string_, the length of _string_ is used in place of _count_.

The **_fstrnset** function is a model-independent (large-model) form of the **strnset** function. The behavior and return value of **_fstrnset** are identical to those of the model-dependent function **strnset**, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strcat, strcmp, strcpy, strset

Example

```
/* STRNSET.C */
#include <string.h>
#include <stdio.h>

void main()
{
   char string[15] = "This is a test";

   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   strnset( string, '*', 4 );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

Description

Scan strings for characters in specified character sets.

#include <string.h>	Required only for function declarations

char *strpbrk(const char ***string1*, const char ***string2* **);**

char _far * _far _fstrpbrk(const char _far ***string1*, const char _far ***string2* **);**

string1	Source string
string2	Character set

Remarks

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.

The **_fstrpbrk** function is a model-independent (large-model) form of the **strpbrk** function. The behavior and return value of **_fstrpbrk** are identical to those of the model-dependent function **strpbrk**, with the exception that all the arguments and return values are far.

Return Value

These functions return a pointer to the first occurrence of any character from *string2* in *string1*. A **NULL** return value indicates that the two string arguments have no characters in common.

Compatibility

strpbrk

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrpbrk

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strchr, strrchr

Example

```
/* STRPBRK.C */
#include <string.h>
#include <stdio.h>

void main()
{
   char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
   char *result;
```

```
    /* Return pointer to first 'a' or 'b' in "string"  */
    printf( "1: %s\n", string );
    result = strpbrk( string, "0123456789" );
    printf( "2: %s\n", result++ );
    result = strpbrk( result, "0123456789" );
    printf( "3: %s\n", result++ );
    result = strpbrk( result, "0123456789" );
    printf( "4: %s\n", result );
}
```

Output

```
1: The 3 men and 2 boys ate 5 pigs

2: 3 men and 2 boys ate 5 pigs

3: 2 boys ate 5 pigs

4: 5 pigs
```

Description Scan a string for the last occurrence of a character.

#include <string.h> Required only for function declarations

char *strrchr(const char ***string*****, int** ***c*** **);**

char _far * _far _fstrrchr(const char _far ***string*****, int** ***c*** **);**

string	Searched string
c	Character to be located

Remarks 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*.)

The **_fstrrchr** function is a model-independent (large-model) form of the **strrchr** function. The behavior and return value of **_fstrrchr** are identical to those of the model-dependent function **strrchr**, with the exception that all the pointer arguments and return values are far pointers.

Return Value These functions return a pointer to the last occurrence of the character in the string. A **NULL** pointer is returned if the given character is not found.

Compatibility **strrchr**

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrrchr

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strchr, strcspn, strncat, strncmp, strncpy, strnicmp, strpbrk, strspn**

Example

```
/* STRCHR.C: This program illustrates searching for a character with
 * strchr (search forward) or strrchr (search backward).
 */
#include <string.h>
#include <stdio.h>
```

```
int  ch = 'r';
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

void main()
{
   char *pdest;
   int result;

   printf( "String to be searched: \n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );
   printf( "Search char:\t%c\n", ch );

   /* Search forward. */
   pdest = strchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );

   /* Search backward. */
   pdest = strrchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tlast %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );

}
```

Output

```
String to be searched:
                The quick brown dog jumps over the lazy fox
                         1         2         3         4         5
                12345678901234567890123456789012345678901234567890

Search char:    r
Result: first r found at position 12

Result: last r found at position 30
```

Description Reverses characters of a string.

#include <string.h> Required only for function declarations

char *strrev(char ***_string_ **);

char _far * _far _fstrrev(char _far ***_string_ **);

string String to be reversed

Remarks The **strrev** function reverses the order of the characters in *string*. The terminating null character (**'\0'**) remains in place.

The **_fstrrev** function is a model-independent (large-model) form of the **strrev** function. The behavior and return value of **_fstrrev** are identical to those of the model-dependent function **strrev**, with the exception that the argument and return value are far pointers.

Return Value These functions return a pointer to the altered string. There is no error return.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **strcpy, strset**

Example

```
/* STRREV.C: This program checks an input string to see whether it is a
 * palindrome: that is, whether it reads the same forward and backward.
 */

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

void main()
{
   char string[100];
   int result;

   printf( "Input a string and I will tell you if it is a palindrome:\n" );
   gets( string );
```

```
   /* Reverse string and compare (ignore case): */
   result = strcmpi( 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 );
}
```

Output

```
Input a string and I will tell you if it is a palindrome:
Able was I ere I saw Elba
The string "Able was I ere I saw Elba" is a palindrome
```

Description Set characters of a string to a character.

#include <string.h> Required only for function declarations

char *strset(char ***_string_, int** _c_ **);**

char _far * _far _fstrset(char _far ***_string_, int** _c_ **);**

string	String to be set
_c_	Character setting

Remarks The **strset** function sets all of the characters of _string_ to _c_, except the terminating null character (**'\0'**).

The **_fstrset** function is a model-independent (large-model) form of the **strset** function. The behavior and return value of **_fstrset** are identical to those of the model-dependent function **strset**, with the exception that the pointer arguments and return value are far pointers.

Return Value These functions return a pointer to the altered string. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **memset, strcat, strcmp, strcpy, strnset**

Example

```
/* STRSET.C */
#include <string.h>
#include <stdio.h>


void main()
{
   char string[] = "Fill the string with something";

   printf( "Before: %s\n", string );
   strset( string, '*' );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: Fill the string with something
After:  ******************************
```

Description

Find the first substring.

#include <string.h>	Required only for function declarations

size_t strspn(const char ***_string1_, const char ***_string2_ **);**

size_t _far _fstrspn(const char _far ***_string1_, const char _far ***_string2_ **);**

string1	Searched string
string2	Character set

Remarks

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.

The **_fstrspn** function is a model-independent (large-model) form of the **strspn** function. The behavior and return value of **_fstrspn** are identical to those of the model-dependent function **strspn**, with the exception that the arguments are far pointers.

Return Value

These functions return an integer value specifying the length of the segment in _string1_ consisting entirely of characters in _string2_.

Compatibility

strspn

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrspn

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strcspn, strncat, strncmp, strncpy, strnicmp, strrchr

Example

```
/* STRSPN.C: This program uses strspn to determine the length of
 * the segment in the string "cabbage" consisting of a's, b's, and c's.
 * In other words, it finds the first non-abc letter.
 */

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

```
void main()
{
   char string[] = "cabbage";
   int  result;

   result = strspn( string, "abc" );
   printf( "The portion of '%s' containing only a, b, or c "
           "is %d bytes long\n", string, result );
}
```

Output

```
The portion of 'cabbage' containing only a, b, or c is 5 bytes long
```

Description

Find a substring.

#include <string.h>	Required only for function declarations

char *strstr(const char ****string1*, const char ****string2***);**

char _far * _far _fstrstr(const char _far ****string1*, const char _far ****string2***);**

string1	Searched string
string2	String to search for

Remarks

The **strstr** function returns a pointer to the first occurrence of *string2* in *string1*.

The **_fstrstr** function is a model-independent (large-model) form of the **strstr** function. The behavior and return value of **_fstrstr** are identical to those of the model-dependent function **strstr**, with the exception that the arguments and return value are far pointers.

Return Value

These functions return either a pointer to the first occurrence of *string2* in *string1*, or **NULL** if they do not find the string.

Compatibility

strstr

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

_fstrstr

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strcspn, strncat, strncmp, strncpy, strnicmp, strpbrk, strrchr, strspn

Example

```
/* STRSTR.C */
#include <string.h>
#include <stdio.h>

char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
```

```
void main()
{
   char *pdest;
   int  result;

   printf( "String to be searched:\n\t%s\n", string );
   printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );

   pdest = strstr( string, str );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "%s found at position %d\n\n", str, result );
   else
      printf( "%s not found\n", str );
}
```

Output

```
String to be searched:
        The quick brown dog jumps over the lazy fox
                 1         2         3         4         5
        12345678901234567890123456789012345678901234567890

lazy found at position 36
```

Description Copies the time to a buffer.

#include <time.h>

char *_strtime(char ***_timestr_ **);

timestr	Time string

Remarks The **_strtime** function copies the current time into the buffer pointed to by *timestr*. The time is 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 6:00 PM.

The buffer must be at least nine bytes long.

Return Value The **_strtime** function returns a pointer to the resulting text string *timestr*.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **asctime, ctime, gmtime, localtime, mktime, time, tzset**

Example

```
/* STRTIME.C */
#include <time.h>
#include <stdio.h>

void main()
{
   char dbuffer [9];
   char tbuffer [9];
```

```
   _strdate( dbuffer );
   printf( "The current date is %s \n", dbuffer );
   _strtime( tbuffer );
   printf( "The current time is %s \n", tbuffer );
}
```

Output

```
The current date is 06/20/89
The current time is 09:33:13
```

Description

Convert strings to a double-precision (**strtod**, **_strtold**), long-integer (**strtol**), or unsigned long-integer (**strtoul**) value.

#include <stdlib.h>

double strtod(const char **nptr*, **char** ***endptr* **);**

long strtol(const char **nptr*, **char** ***endptr*, **int** *base* **);**

long double _strtold(const char **nptr*, **char** ***endptr* **);**

unsigned long strtoul(const char **nptr*, **char** ***endptr*, **int** *base* **);**

nptr	String to convert
endptr	End of scan
base	Number base to use

Remarks

The **strtod**, **_strtold**, **strtol**, and **strtoul** functions convert a character string to a double-precision value, a long-double 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. If the **strtod** or **_strtold** function appears in a compact-, large-, or huge-model program, *nptr* can be a maximum of 100 characters.

These functions stop reading the string at the first character they cannot recognize as part of a number. This may be the null character (**'\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 **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion could be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

The **strtod** and **_strtold** functions expect *nptr* to point to a string with the following form:

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

The first character that does not 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 not 'x' or 'X', then the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. 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' through '9', then the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted.

The **strtoul** function allows a plus (+) or minus (–) sign prefix; a leading minus sign indicates that the return value is negated.

Return Value

The **strtod** and **_strtold** functions return the value of the floating-point number, except when the representation would cause an overflow, in which case it returns **±HUGE_VAL**. The functions return 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 function 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 four functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

Compatibility

strtod, strtol, _strtold

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

strtoul

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

atof, atol

Example

```
/* STRTOD.C: This program uses strtod to convert a string to a
 * double-precision value; strtol to convert a string to long
 * integer values; and strtoul to convert a string to unsigned
 * long-integer values.
 */

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

```
void main()
{
   char   *string, *stopstring;
   double x;
   long   l;
   int    base;
   unsigned long ul;

   string = "3.1415926This stopped it";
   x = strtod( string, &stopstring );
   printf( "string = %s\n", string );
   printf("   strtod = %f\n", x );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "-10110134932This stopped it";
   l = strtol( string, &stopstring, 10 );
   printf( "string = %s\n", string );
   printf("   strtol = %ld\n", l );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "10110134932";
   printf( "string = %s\n", string );
   /* Convert string using base 2, 4, and 8: */
   for( base = 2; base <= 8; base *= 2 )
   {
      /* Convert the string: */
      ul = strtoul( string, &stopstring, base );
      printf( "   strtol = %ld (base %d)\n", ul, base );
      printf( "   Stopped scan at: %s\n", stopstring );
   }
}
```

Output

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at: This stopped it

string = -10110134932This stopped it
   strtol = -2147483647
   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
```

Description

Find the next token in a string.

#include <string.h>	Required only for function declarations

char *strtok(char ***string1*, const char** ***string2*);**

char _far * _far _fstrtok(char _far ***string1*, const char _far** ***string2*);**

string1	String containing token(s)
string2	Set of delimiter characters

Remarks

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 can be 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.

The **_fstrtok** function is a model-independent (large-model) form of the **strtok** function. The behavior and return value of **_fstrtok** are identical to those of the model-dependent function **strtok**, with the exception that the arguments and return value are far pointers.

Note that calls to these functions 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.

Compatibility

strtok

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

_fstrtok

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

strcspn, strspn

Example

```
/* STRTOK.C: In this program, a loop uses strtok to print all the tokens
 * (separated by commas or blanks) in the string named "string".
 */

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

char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";
char *token;

void main()
{

   printf( "%s\n\nTokens:\n", string );

   /* Establish string and get the first token: */
   token = strtok( string, seps );
   while( token != NULL )
   {
      /* While there are tokens in "string" */
      printf( " %s\n", token );
      /* Get next token: */
      token = strtok( NULL, seps );
   }
}
```

Output

```
A string        of ,,tokens
and some  more tokens

Tokens:
 A
 string
 of
 tokens
 and
 some
 more
 tokens
```

Description Convert a string to uppercase.

#include <string.h> Required only for function declarations

char *strupr(char ***string* **);

char _far * _far _fstrupr(char _far ***string* **);

string	String to be capitalized

Remarks These functions convert any lowercase letters in the string to uppercase. Other characters are not affected.

The **_fstrupr** function is a model-independent (large-model) form of the **strupr** function. The behavior and return value of **_fstrupr** are identical to those of the model-dependent function **strupr**, with the exception that the argument and return value are far pointers.

Return Value These functions return a pointer to the converted string. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **strlwr**

Example

```
/* STRLWR.C: This program uses strlwr and strupr to create
 * uppercase and lowercase copies of a mixed-case string.
 */

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

void main()
{
   char string[100] = "The String to End All Strings!";
   char *copy1, *copy2;

   copy1 = strlwr( strdup( string ) );
   copy2 = strupr( strdup( string ) );
   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
}
```

Output

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

Description

Transforms a string based on locale-specific information.

#include <string.h>	Required only for function declarations

size_t strxfrm(char ***string1*****, const char** ***string2*****, size_t** *count* **);**

string1	String to which transformed version of *string2* is returned
string2	String to transform
count	Maximum number of characters to be placed in *string1*

Remarks

The **strxfrm** function transforms the string pointed to by *string2* into a new form that is stored in *string1*. No more than *count* characters (including the null character) are transformed and placed into the resulting string.

The transformation is made using the information in the locale-specific **LC_COLLATE** macro.

After the transformation, a call to **strcmp** with the two transformed strings will yield identical results to a call to **strcoll** applied to the original two strings.

The value of the following expression is the size of the array needed to hold the transformation of the source string:

```
1 + strxfrm( NULL, string, 0 )
```

Currently, the C libraries support the "C" locale only; thus **strxfrm** is equivalent to the following:

```
strncpy( _string1, _string2, _count );
return( strlen( _string2 ) );
```

Return Value

The **strxfrm** function returns the length of the transformed string, not counting the terminating null character. If the return value is greater than or equal to *count*, the contents of *string1* are unpredictable.

Compatibility

■ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

localeconv, setlocale, strncmp

Description Swaps bytes.

#include <stdlib.h>	Required only for function declarations

void swab(char ***src*****, char** ***dest*****, int** ***n*** **);**

src	Data to be copied and swapped
dest	Storage location for swapped data
n	Number of bytes to be copied and swapped

Remarks The **swab** function copies *n* bytes from *src*, swaps each pair of adjacent bytes, and stores the result at *dest*. 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 None.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

Example

```
/* SWAB.C */
#include <stdlib.h>
#include <stdio.h>

char from[] = "BADCFEHGJILKNMPORQTSVUXWZY";
char to[] =   "..........................";

void main()
{
   printf( "Before:\t%s\n\t%s\n\n", from, to );
   swab( from, to, sizeof( from ) );
   printf( "After:\t%s\n\t%s\n\n", from, to );
}
```

Output

```
Before: BADCFEHGJILKNMPORQTSVUXWZY
        ..........................

After:  BADCFEHGJILKNMPORQTSVUXWZY
        ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Description Executes a command.

#include <process.h>	Required only for function declarations
#include <stdlib.h>	Use STDLIB.H for ANSI compatibility

int system(const char ****command* **);**

command	Command to be executed

Remarks The **system** function passes *command* to the command interpreter, which executes the string as an operating-system command. The **system** function refers to the COMSPEC and PATH environment variables that locate the command-interpreter file (the file named COMMAND.COM in DOS or CMD.EXE in OS/2). If *command* is a pointer to an empty string, the function simply checks to see whether or not the command interpreter exists.

Return Value If *command* is **NULL** and the command interpreter is found, the function returns a nonzero value. If the command interpreter is not found, it returns the value 0 and sets **errno** to **ENOENT**. If *command* is not **NULL**, the **system** function returns the value 0 if the command interpreter is successfully started. Under OS/2, the **system** function returns the exit status from the command interpreter.

A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
E2BIG	In DOS, the argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K. In OS/2, the combined argument list and space required for environment information exceed 32K.
ENOENT	The command interpreter cannot be found.
ENOEXEC	The command-interpreter 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also exec functions, exit, _exit, spawn functions

Example

```
/* SYSTEM.C: This program uses system to TYPE its source file. */

#include <process.h>

void main()
{
   system( "type system.c" );
}
```

Output

```
/* SYSTEM.C: This program uses system to TYPE its source file. */

#include <process.h>

void main()
{
   system( "type system.c" );
}
```

Description

Calculate the tangent (**tan** and **tanl**) and hyperbolic tangent (**tanh** and **tanhl**).

#include <math.h>

double tan(double *x* **);**

double tanh(double *x* **);**

long double tanl(long double *x* **);**

long double tanhl(long double *x* **);**

x	Angle in radians

Remarks

The **tan** functions return the tangent or hyperbolic tangent of their arguments. The list below describes the differences between the various tangent functions:

Function	Description
tan	Calculates tangent of *x*
tanh	Calculates hyperbolic tangent of *x*
tanl	Calculates tangent of *x* (80-bit version)
tanhl	Calculates hyperbolic tangent of *x* (80-bit version)

The **tanl** and **tanhl** functions are the 80-bit counterparts and use an 80-bit, 10-byte coprocessor form of arguments and return values. See the reference page on the long double functions for more details on this data type.

Return Value

The **tan** function returns the tangent of *x*. If *x* is large, a partial loss of significance in the result may occur; in this case, **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**.

Compatibility

tan, tanh

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

tanl, tanhl

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **acos** functions, **asin** functions, **atan** functions, **cos** functions, **sin** functions

Example

```
/* TAN.C:  This program displays the tangent of pi / 4 and the hyperbolic
 * tangent of the result.
 */

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

void main()
{
   double pi = 3.1415926535;
   double x, y;

   x = tan( pi / 4 );
   y = tanh( x );
   printf( "tan( %f ) = %f\n", x, y );
   printf( "tanh( %f ) = %f\n", y, x );
}
```

Output

```
tan( 1.000000 ) = 0.761594
tanh( 0.761594 ) = 1.000000
```

Description Gets the position of the file pointer.

#include <io.h>	Required only for function declarations

long tell(int *handle* **);**

handle	Handle referring to open file

Remarks The **tell** function gets the current position of the file pointer (if any) associated with the *handle* argument. 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.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **ftell, lseek**

Example

```
/* TELL.C:  This program uses tell to tell the file pointer position
 * after a file read.
 */

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

void main()
{
   int  fh;
   long position;
   char buffer[500];

   if( (fh = open( "tell.c", O_RDONLY )) != -1 )
   {
      if( read( fh, buffer, 500 ) > 0 )
         printf( "Current file position is: %d\n", tell( fh ) );
```

```
      close( fh );
   }
}
```

Output

```
Current file position is: 425
```

Description

Create temporary file names.

#include <stdio.h>

char *tempnam(char **dir*, **char** **prefix* **);**

char *tmpnam(char **string* **);**

string	Pointer to temporary name
dir	Target directory to be used if TMP not defined
prefix	File-name prefix

Remarks

The **tmpnam** function generates a temporary file name that can be used to open a temporary file without overwriting an existing 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 destroy this value. If *string* is not **NULL**, it is assumed to point to an array of at least **L_tmpnam** bytes (the value of **L_tmpnam** is defined in STDIO.H). 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 entry **P_tmpdir** in the file 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 program to create a temporary file name for use in another directory. This file name will be different from that of any existing file. The *prefix* argument is the prefix to the file name. The **tempnam** function uses **malloc** to allocate space for the file name; the program 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 the search through the locations listed above 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 a file with that name already exists, **tmpnam** and **tempnam** return the value **NULL.**

Compatibility

tmpnam

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

tempnam

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

tmpfile

Example

```
/* TEMPNAM.C: This program uses tmpnam to create a unique file name in
 * the current working directory, then uses tempnam to create a unique
 * file name with a prefix of stq.
 */

#include <stdio.h>

void 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 in temporary directory with the
    * prefix "stq". The actual destination directory may vary depending
    * on the state of the TMP environment variable and the global variable
    * P_tmpdir.
    */
   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" );
}
```

Output

```
\2 is safe to use as a temporary file.
C:\TMP\stq2 is safe to use as a temporary file.
```

Description

Gets the system time.

#include <time.h> Required only for function declarations

time_t time(time_t ***timer* **);

timer Storage location for time

Remarks

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 adjusted according to the **timezone** system variable, which is explained in the **tzset** reference section.

The return value is stored in the location given by *timer*. 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

asctime, ftime, gmtime, localtime, tzset, utime

Example

```
/* CTIME.C: This program gets the current time in time_t form, then uses
 * ctime to display the time in string form.
 */

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

void main()
{
   time_t ltime;

   time( &ltime );
   printf( "The time is %s\n", ctime( &ltime ) );
}
```

Output

```
The time is Thu Jun 15 16:08:18 1989
```

Description

Creates a temporary file.

#include <stdio.h>

FILE *tmpfile(void);

Remarks

The **tmpfile** function creates a temporary file and returns a pointer to that stream. 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.

Return Value

If successful, the **tmpfile** function returns a stream pointer. Otherwise, it returns a **NULL** pointer.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

rmtmp, tempnam, tmpnam

Example

```
/* TMPFILE.C: This program uses tmpfile to create a temporary file,
 * then deletes this file with rmtmp.
 */

#include <stdio.h>

void main()
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;

   /* Create temporary files. */
   for( i = 1; i <= 10; i++ )
   {
      if( (stream = tmpfile()) == NULL )
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }

   /* Remove temporary files. */
```

```
   printf( "%d temporary files deleted\n", rmtmp() );
}
```

Output

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
Temporary file 4 was created
Temporary file 5 was created
Temporary file 6 was created
Temporary file 7 was created
Temporary file 8 was created
Temporary file 9 was created
Temporary file 10 was created
10 temporary files deleted
```

Description Convert characters.

#include <ctype.h>

int toascii(int *c* **);**

int tolower(int *c* **);**

int _tolower(int *c* **);**

int toupper(int *c* **);**

int _toupper(int *c* **);**

c	Character to be converted

Remarks The **toascii, tolower, _tolower, toupper**, and **_toupper** routines convert a single character, as described below:

Function	Description
toascii	Converts *c* to ASCII character
tolower	Converts *c* to lowercase if appropriate
_tolower	Converts *c* to lowercase
toupper	Converts *c* to uppercase if appropriate
_toupper	Converts *c* to uppercase

The **toascii** routine 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** and **_tolower** routines convert *c* to lowercase if *c* represents an uppercase letter. Otherwise, *c* is unchanged. The **_tolower** routine 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** and **_toupper** routines convert *c* to uppercase if *c* represents a lowercase letter. Otherwise, *c* is unchanged. The **_toupper** routine 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 that these routines are implemented both as functions and as macros. To conform with the ANSI specification, the **tolower** and **toupper** routines are also implemented as functions. 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.

If the -Za compile option is used, the macro form of **toupper** or **tolower** is not used because it evaluates its argument more than once. Since the arguments are evaluated more than once, arguments with side effects would produce potentially bad results.

Return Value

The **toascii, tolower, _tolower, toupper**, and **_toupper** routines return the converted character *c*. There is no error return.

Compatibility

toascii, _tolower, _toupper

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

tolower, toupper

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

is functions

Example

```
/* TOUPPER.C: 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.
 */

#include <conio.h>
#include <ctype.h>
#include <string.h>

char msg[] = "Some of THESE letters are Capitals\r\n";
char *p;

void main()
{
   cputs( msg );
```

```
   /* Reverse case of message. */
   for( p = msg; p < msg + strlen( msg ); p++ )
   {
      if( islower( *p ) )
         putch( _toupper( *p ) );
      else if( isupper( *p ) )
         putch( _tolower( *p ) );
      else
         putch( *p );
   }
}
```

Output

```
Some of THESE letters are Capitals
sOME OF these LETTERS ARE cAPITALS
```

Description

Sets time environment variables.

#include <time.h>	Required only for function declarations

void tzset(void);

int daylight; **long timezone;** **char *tzname[2]**	Global variables set by function

Remarks

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.

If the TZ value is not currently set, the default is PST8PDT, which corresponds to the Pacific time zone.

Based on the TZ environment variable value, the following values are assigned to the variables **daylight**, **timezone**, and **tzname** when **tzset** is called:

Variable	Value
daylight	Nonzero value if a daylight-saving-time zone is specified in the TZ setting; otherwise, 0
timezone	Difference in seconds between GMT and local time
tzname[0]	String value of the three-letter time-zone name from the TZ setting
tzname[1]	String value of the daylight-saving-time zone, or an empty string if the daylight-saving-time zone is omitted from the TZ setting

The default for **daylight** is 1; for **timezone**, 28,800; for **tzname**[0], PST; and for **tzname**[1], PDT. This corresponds to "PST8PDT."

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.

Return Value None.

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **asctime, ftime, gmtime, localtime, time**

Example

```
/* TZSET.C: 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
 * global variables named daylight, timezone, and tzname.
 */

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

void main()
{

   if( putenv( "TZ=EST5EDT" ) == -1 )
   {
      printf( "Unable to set TZ\n" );
      exit( 1 );
   }
   else
   {
      tzset();
      printf( "daylight = %d\n", daylight );
      printf( "timezone = %ld\n", timezone );
      printf( "tzname[0] = %s\n", tzname[0] );
   }
}
```

Output

```
daylight = 1
timezone = 18000
tzname[0] = EST
```

Description Converts an unsigned long integer to a string.

#include <stdlib.h> Required only for function declarations

char *ultoa(unsigned long *value*, **char ***string*, **int** *radix* **);**

value	Number to be converted
string	String result
radix	Base of *value*

Remarks The **ultoa** function converts *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.

Compatibility □ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also **itoa, ltoa**

Example

```
/* ITOA.C: This program converts integers of various sizes to strings
 * in various radixes.
 */

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

void main()
{
   char buffer[20];
   int  i = 3445;
   long l = -344115L;
   unsigned long ul = 1234567890UL;

   itoa( i, buffer, 10 );
   printf( "String of integer %d (radix 10): %s\n", i, buffer );
   itoa( i, buffer, 16 );
   printf( "String of integer %d (radix 16): 0x%s\n", i, buffer );
   itoa( i, buffer, 2  );
   printf( "String of integer %d (radix 2): %s\n", i, buffer );
```

```
   ltoa( l, buffer, 16 );
   printf( "String of long int %ld (radix 16): 0x%s\n", l, buffer );

   ultoa( ul, buffer, 16 );
   printf( "String of unsigned long %lu (radix 16): 0x%s\n", ul, buffer );
}
```

Output

```
String of integer 3445 (radix 10): 3445
String of integer 3445 (radix 16): 0xd75
String of integer 3445 (radix 2): 110101110101
String of long int -344115 (radix 16): 0xfffabfcd
String of unsigned long 1234567890 (radix 16): 0x499602d2
```

Description

Sets the default file-permission mask.

#include <sys\types.h>

#include <sys\stat.h>

#include <io.h> Required only for function declarations

int umask(int *pmode* **);**

pmode Default permission setting

Remarks

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_IREAD** and **S_IWRITE,** 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_IREAD	Reading not allowed (file is write-only)
S_IWRITE	Writing not allowed (file is read-only)

For example, if the write bit is set in the mask, any new files will be read-only.

Note that under DOS and OS/2, 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 modes.

Return Value

The **umask** function returns the previous value of *pmode*. There is no error return.

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also chmod, creat, mkdir, open

Example

```
/* UMASK.C: 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.
 */

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

void main()
{
   int oldmask;

   /* Create read-only files: */
   oldmask = umask( S_IWRITE );
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

Output

```
Oldmask = 0x0000
```

Description

Pushes a character back onto the stream.

#include <stdio.h>

int ungetc(int *c*, **FILE** **stream* **);**

c	Character to be pushed
stream	Pointer to **FILE** structure

Remarks

The **ungetc** function pushes the character *c* back onto *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.

Results are unpredictable if the **ungetc** function is called twice without a read operation between the two calls. After a call to the **fscanf** function, a call to **ungetc** may fail unless another read operation (such as the **getc** function) has been performed. This is because the **fscanf** function itself calls the **ungetc** function.

Return Value

The **ungetc** function returns the character argument *c*. The return value **EOF** indicates a failure to push back the specified character.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

getc, getchar, putc, putchar

Example

```
/* UNGETC.C: 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.
 */
```

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

void main()
{
   int ch;
   int result = 0;

   printf( "Enter an integer: " );

   /* Read in and convert number: */
   while( ((ch = getchar()) != EOF) && isdigit( ch ) )
      result = result * 10 + ch - '0';       /* Use digit. */
   if( ch != EOF )
      ungetc( ch, stdin );                   /* Put non-digit back. */
   printf( "Number = %d\nNext character in stream = '%c'\n",
           result, getchar() );
}
```

Output

```
Enter an integer: 521a
Number = 521
Next character in stream = 'a'
```

Description

Pushes back the last character read from the console.

#include <conio.h> Required only for function declarations

int ungetch(int *c* **);**

c Character to be pushed

Remarks

The **ungetch** function pushes the character *c* back to the console, causing *c* to be the next character read by **getch** or **getche.** 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.

Compatibility

□ ANSI ■ DOS ■ OS/2 □ UNIX □ XENIX

See Also

cscanf, getch, getche

Example

```
/* UNGETCH.C: In this program, a white-space delimited token is read
 * from the keyboard. When the program encounters a delimiter,
 * it uses ungetch to replace the character in the keyboard buffer.
 */

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

void main()
{
   char buffer[100];
   int count = 0;
   int ch;
```

```
   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 );
}
```

Output

```
White
token = White
```

Description

Deletes a file.

#include <io.h>	Required only for function declarations
#include <stdio.h>	Use either IO.H or STDIO.H

int unlink(const char ***filename*** **);**

filename	Name of file to remove

Remarks

The **unlink** function deletes the file specified by *filename*.

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 read-only file
ENOENT	File or path name not found, or path name specified a directory

Compatibility

□ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

close, remove

Example

```
/* UNLINK.C: This program uses unlink to delete UNLINK.OBJ. */

#include <stdio.h>

void main()
{
   if( unlink( "unlink.obj" ) == -1 )
      perror( "Could not delete 'UNLINK.OBJ'" );
   else
      printf( "Deleted 'UNLINK.OBJ'\n" );
}
```

Output

```
Deleted 'UNLINK.OBJ'
```

Description Frees memory used by fonts.

#include <graph.h>

void _far _unregisterfonts(void);

Remarks The **_unregisterfonts** function frees memory previously allocated and used by the **_registerfonts** function. The **_unregisterfonts** function removes the header information for all fonts and unloads the currently selected font data from memory.

Any attempt to use the **_setfont** or **_outgtext** function after calling **_unregisterfonts** results in an error.

Return Value None.

Compatibility □ ANSI ■ DOS □ OS/2 □ UNIX □ XENIX

See Also **_getfontinfo, _getgtextextent, _outgtext, _registerfonts, _setfont**

Example See the example for **_outgtext**.

Description Sets the file modification time.

#include <sys\types.h>

#include <sys\utime.h>

int utime(char ***filename*****, struct utimbuf** ****times*** **);**

filename	File name
times	Pointer to stored time values

Remarks The **utime** function sets the modification time for the file specified by *filename*. 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, only the modification time is set under DOS and OS/2 . 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

Compatibility □ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **asctime, ctime, fstat, ftime, gmtime, localtime, stat, time**

Example

```
/* UTIME.C: This program uses utime to set the file-modification time to
 * the current time.
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys\types.h>
#include <sys\utime.h>

void main()
{
   /* Show file time before and after. */
   system( "dir utime.c" );
   if( utime( "utime.c", NULL ) == -1 )
      perror( "utime failed\n" );
   else
      printf( "File time modified\n" );
   system( "dir utime.c" );
}
```

Output

```
 The volume label in drive C is OS2.
 Directory of C:\LIBREF

UTIME    C        397   6-20-89   2:11p
     1 File(s)   12974080 bytes free
File time modified

 The volume label in drive C is OS2.
 Directory of C:\LIBREF

UTIME    C        397   6-20-89   2:12p
     1 File(s)   12974080 bytes free
```

Description

Access variable-argument lists.

#include <stdarg.h>	Required for ANSI compatibility
#include <varargs.h>	Required for UNIX V compatibility
#include <stdio.h>	

type **va_arg(va_list** *arg_ptr*, *type* **);**

void va_end(va_list *arg_ptr* **);**

void va_start(va_list *arg_ptr* **);**	UNIX version
void va_start(va_list *arg_ptr*, *prev_param* **);**	ANSI

arg_ptr	Pointer to list of arguments
prev_param	Parameter preceding first optional argument (ANSI only)
type	Type of argument to be retrieved

Remarks

The **va_arg**, **va_end**, and **va_start** 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. The macros are listed below:

Macro	Description
va_alist	Name of parameter to called function (UNIX version only)
va_arg	Macro to retrieve current argument
va_dcl	Declaration of **va_alist** (UNIX version only)
va_end	Macro to reset *arg_ptr*
va_list	The **typedef** for the pointer to list of arguments
va_start	Macro to set *arg_ptr* to beginning of list of optional arguments (UNIX version only)

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 macros in STDARG.H or VARARGS.H, 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.

Compatibility ■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **vfprintf, vprintf, vsprintf**

Example

```
/* VA.C: The program below illustrates passing a variable number of arguments
 * using the following macros:
 *      va_start            va_arg              va_end
 *      va_list             va_decl (UNIX only)
 */

#include <stdio.h>
#define ANSI            /* Comment out for UNIX version     */
#ifdef ANSI             /* ANSI compatible version          */
#include <stdarg.h>
int average( int first, ... );
#else                   /* UNIX compatible version          */
#include <varargs.h>
int average( va_list );
#endif

void main()
{
   /* Call with 3 integers (-1 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );

   /* Call with 4 integers. */
   printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );

   /* Call with just -1 terminator. */
   printf( "Average is: %d\n", average( -1 ) );
}

/* Returns the average of a variable list of integers. */
#ifdef ANSI             /* ANSI compatible version    */
int average( int first, ... )
{
   int count = 0, sum = 0, i = first;
   va_list marker;
```

```
   va_start( marker, first );      /* Initialize variable arguments. */
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);
   }
   va_end( marker );               /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#else       /* UNIX compatible version must use old-style definition. */
int average( va_alist )
va_dcl
{
   int i, count, sum;
   va_list marker;

   va_start( marker );             /* Initialize variable arguments. */
   for( sum = count = 0; (i = va_arg( marker, int)) != -1;  count++ )
      sum += i;
   va_end( marker );               /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#endif
```

Output

```
Average is: 3
Average is: 8
Average is: 0
```

Description

Write formatted output using a pointer to a list of arguments.

#include <stdio.h>	
#include <varargs.h>	Required for compatibility with UNIX System V
#include <stdarg.h>	Required for compatibility with the ANSI C standard

int vfprintf(FILE **stream*, **const char** **format*, **va_list** *argptr* **);**

int vprintf(const char **format*, **va_list** *argptr* **);**

int vsprintf(char **buffer*, **const char** **format*, **va_list** *argptr* **);**

stream	Pointer to **FILE** structure
format	Format control
argptr	Pointer to list of arguments
buffer	Storage location for output

Remarks

The **vfprintf**, **vprintf**, and **vsprintf** functions format data and output data to the file specified by *stream*, to standard output, and to the memory pointed to by *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* argument has the same form and function as the *format* argument for the **printf** function; see **printf** for a description of *format*.

The *argptr* parameter has type **va_list**, which is defined in the include files 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 return value for **vprintf** and **vsprintf** 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.

Compatibility

■ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also fprintf, printf, sprintf, va_arg, va_end, va_start

Example

```
/* VPRINTF.C shows how to use vprintf functions to write new versions
 * of printf. The vsprintf function is used in the example.
 */

#include <stdio.h>
#include <graph.h>
#include <string.h>
#include <stdarg.h>
#include <malloc.h>

int wprintf( short row, short col, short clr, long bclr, char *fmt, ... );

void main()
{
   short fgd = 0;
   long  bgd = 0L;

   _clearscreen( _GCLEARSCREEN );
   _outtext( "Color text example:\n\n" );

   /* Loop through 8 background colors. */
   for( bgd = 0L; bgd < 8; bgd++ )
   {
      wprintf( (int)bgd + 3, 1, 7, bgd, "Back: %d Fore:", bgd );

      /* Loop through 16 foreground colors. */
      for( fgd = 0; fgd < 16; fgd++ )
         wprintf( -1, -1, fgd, -1L, " %2d ", fgd );
   }
}

/* Full-screen window version of printf that takes row, column, textcolor,
 * and background color as its first arguments, followed by normal printf
 * format strings (except that \t is not handled). You can specify -1 for
 * any of the first arguments to use the current value. The function returns
 * the number of characters printed, or a negative number for errors.
 */
int wprintf( short row, short col, short clr, long bclr, char *fmt, ... )
{
   struct  rccoord tmppos;
   short   ret, size;
   va_list marker;
   char    *buffer;
```

```
    /* It's probably safe to use a buffer length of 512 bytes or five times
     * the length of the format string.
     */
    size = strlen( fmt );
    size = (size > 512) ? 512 : size * 5;
    if( (buffer = (char *)malloc( size )) == NULL )
       return -1;

    /* Set text position. */
    tmppos = _gettextposition();
    if( row < 1 )
       row = tmppos.row;
    if( col < 1 )
       col = tmppos.col;
    _settextposition( row, col );

    /* Set foreground and background colors. */
    if( clr >= 0 )
       _settextcolor( clr );
    if( bclr >= 0 )
       _setbkcolor( bclr );

    /* Write text to a string and output the string. */
    va_start( marker, fmt );
    ret = vsprintf( buffer, fmt, marker );
    va_end( marker );
    _outtext( buffer );
    free( buffer );
    return ret;
}
```

Description

Suspends the calling process.

#include <process.h>

int wait(int **termstat* **);**

termstat	Termination-status word and return code for terminated child process

Remarks

The **wait** function suspends the calling process until any of the caller's immediate child processes terminate. If all of the caller's children have terminated before it calls the **wait** function, the function returns immediately.

If not **NULL**, *termstat* points to a buffer containing a termination-status word and return code for the child process. The status word indicates whether or not the child process ended normally by calling the OS/2 **DosExit** function. Supply **NULL** if you do not need the child's termination-status word.

If the child process did terminate normally, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents
Low order	0
High order	Low-order byte of the result code passed by the child process to **DosExit**. The **DosExit** function is called if the child process called **exit** or **_exit**, if it returned from **main**, or if it reached the end of **main**. The low-order byte of the result code is either the low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or a random value if the child process reached the end of **main**.

Note that the OS/2 **DosExit** function allows programs to return a 16-bit result code. However, the **wait** and **cwait** functions will return only the low-order byte of that result code.

If the child process terminated for any other reason, the high-order and low-order bytes of the termination-status word are as follows:

Byte	Contents	
Low order	Termination code from **DosWait**:	
	Code	**Meaning**
	1	Hard-error abort
	2	Trap operation
	3	**SIGTERM** signal not intercepted
High order	0	

Return Value

If **wait** returns after normal termination of a child process, it returns the child's process ID.

If **wait** returns after abnormal termination of a child process, it returns the number –1 and sets **errno** to **EINTR**.

Otherwise, **wait** returns –1 immediately and sets **errno** to **ECHILD**, indicating that no child processes exist for the calling process.

Compatibility

□ ANSI □ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also

cwait, exit, _exit

Example

```
/* WAIT.C: This program launches several child processes and waits for
 * the first to finish.
 */

#define INCL_NOPM
#define INCL_NOCOMMON
#define INCL_DOSPROCESS
#include <os2.h>        /* DosSleep   */
#include <process.h>    /* wait       */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

```
/* Macro to get a random integer within a specified range */
#define getrandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))

struct  CHILD
{
    int     pid;
    char    name[10];
} child[4] = { { 0, "Ann" }, { 0, "Beth"  }, { 0, "Carl" }, { 0, "Dave" } };

void main( int argc, char *argv[] )
{
    int     termstat, pid, c, tmp;

    srand( (unsigned)time( NULL ) );               /* Seed randomizer */
    /* If no arguments, this is the parent. */
    if( argc == 1 )
    {

        /* Spawn children in random order with a random delay. */
        tmp = getrandom( 0, 3 );
        for( c = tmp; c < tmp + 4; c++ )
            child[c % 4].pid = spawnl( P_NOWAIT, argv[0], argv[0],
                                       child[c % 4].name, NULL );

        /* Wait for the first children. Only get ID of first. */
        printf( "Who's first?\n" );
        pid = wait( &termstat );
        for( c = 0; c < 3; c++ )
            wait( &termstat );

        /* Check IDs to see who was first. */
        for( c = 0; c < 4; c++ )
            if( pid == child[c].pid )
                printf( "%s was first\n\n", child[c].name );
    }

    /* If there are arguments, this must be a child. */
    else
    {
        /* Delay for random time. */
        srand( (unsigned)time( NULL ) * argv[1][0] );
        DosSleep( getrandom( 1, 5) * 1000L );
        printf( "Hi, dad. It's %s.\n", argv[1] );
    }
}
```

Output

```
Who's first?
Hi, dad. It's Carl.
Hi, dad. It's Ann.
Hi, dad. It's Beth.
Hi, dad. It's Dave.
Carl was first
```

Description Controls word wrap.

#include <graph.h>

short _far _wrapon(short *option* **);**

option Wrap condition

Remarks The **_wrapon** function controls whether text output with the **_outtext** function 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

Note that this function does not affect the output of presentation-graphics routines or font routines.

Return Value The function returns the previous value of *option*. There is no error return.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ☐ UNIX ☐ XENIX

See Also **_outtext, _settextwindow**

Example

```
/* WRAPON.C */

#include <conio.h>
#include <graph.h>

void main()
{
   _wrapon( _GWRAPON );
   while( !kbhit() )
      _outtext( "Wrap on!   " );
   getch();
   _outtext( "\n\n" );

   _wrapon( _GWRAPOFF );
   while( !kbhit() )
```

```
      _outtext( "Wrap off!  " );
   getch();
   _outtext( "\n\n" );
}
```

Output

```
Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap
 on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!
  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wr
ap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap on!  Wrap o
n!  Wrap on!  Wrap on!

Wrap off! Wrap off! Wrap off! Wrap off! Wrap off! Wrap off! Wrap off! Wrap
```

Description

Writes data to a file.

#include <io.h>	Required only for function declarations

int write(int *handle*, **void** **buffer*, **unsigned int** *count* **);**

handle	Handle referring to open file
buffer	Data to be written
count	Number of bytes

Remarks

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 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 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.

When writing 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.

Compatibility ☐ ANSI ■ DOS ■ OS/2 ■ UNIX ■ XENIX

See Also **fwrite, open, read**

Example

```
/* WRITE.C: This program opens a file for output and uses write to
 * write some bytes to the file.
 */

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

char buffer[] = "This is a test of 'write' function";

void main()
{
   int fh;
   unsigned byteswritten;

   if( (fh = open( "write.o", O_RDWR | O_CREAT, S_IREAD | S_IWRITE )) != -1 )
   {
      if( (byteswritten = write( fh, buffer, sizeof( buffer ) )) == -1 )
         perror( "Write failed" );
      else
         printf( "Wrote %u bytes to file\n", byteswritten );

      close( fh );
   }
}
```

Output

```
Wrote 35 bytes to file
```

Index

A

abort, 51, 76
abs, 78
Absolute value
    abs, 78
    cabs, 134
    cabsl, 134
    fabs, 263
    fabsl, 263
    labs, 441
access, 24, 80
Access mode, 269, 295, 315, 326
acos, 46, 82
acosl, 46, 82
alloca, 48, 84
Allocation. *See* Memory allocation
_amblksiz variable, 63
Appending
    constants, 523, 704
    streams, 295, 315, 326
_arc, _arc_w, _arc_wxy
    description, 86
    use, 30
Arccosine function, 82
Arcsine function, 90
Arctangent function, 94
Arguments
    singularity, 480
    type checking, vi
    variable-length number, 62, 817
asctime, 60, 88
asin, 46, 90
asinl, 46, 90
assert, 92
Assertions, 92
atan, atan2, 46, 94
atanl, atan2l, 46, 94
atexit, 51, 96
atof, atoi, atol, _atold, 22, 98
Attributes, 29

B

_bcalloc, 136
bdos, 57, 101
_beginthread, 103
Bessel functions
    described, 48, 107
    j0,j1,jn, 107
    _j0l,_j1l,_jnl, 107
    y0,y1,yn, 107
    _y0l,_y1l,_ynl, 107
_bexpand, 260
_bfree, 310
_bfreeseg, 110
_bheapadd, 406
_bheapchk, 409
_bheapmin, 411
_bheapseg, 112
_bheapset, 412
_bheapwalk, 415
Binary
    format, conversion to IEEE
        double precision, 181
    int
        reading, 389
        writing, 597
    mode
        _fmode, 66
        fdopen, 269
        fopen, 296
        freopen, 315–316
        _fsopen, 327
        open, 523
        setmode, 664
        sopen, 704
        vs. text mode, 35
    search, 132, 447, 466
BINMODE.OBJ, 66
_bios_disk, 57, 115
_bios_equiplist, 57, 119
_bios_keybrd, 57, 121
_bios_memsize, 57, 124
_bios_printer, 57, 125
_bios_serialcom, 57, 127
_bios_timeofday, 57, 130
_bmalloc, 476
_bmsize, 519
Bold type, use of, ix
Brackets, double, use of, ix
_brealloc, 607
bsearch, 55, 132
Buffer manipulation
    _fmemccpy, 487
    _fmemchr, 489
    _fmemcmp, 491
    _fmemcpy, 494

Buffer manipulation (*continued*)
  _fmemicmp, 497
  _fmemmove, 501
  _fmemset, 504
  memccpy, 487
  memchr, 489
  memcmp, 491
  memcpy, 494
  memicmp, 497
  memmove, 501
  memset, 504
Buffering
  described, 37
  preopened streams, 40
  using, 40
Buffers
  assigning, 648
  comparing, 491, 497
  copying, 487, 494
  flushing, 276, 292
  searching, 489
  setting characters, 504
BUFSIZ constant, 37
Byte order, swapping, 783

C

cabs, 46, 134
cabsl, 46, 134
calloc, 48, 136
Carry flag
  bdos, 101
  int86, 426
  int86x, 428
  intdos, 430
  intdosx, 432
Case in file names, 9
ceil, 46, 138
Ceiling function, 138
ceill, 46, 138
_cexit, _c_exit, 140
cgets, 44, 141
_chain_intr, 57, 60, 143
Character classification and conversion functions
  include files, 22
  isalnum, 21, 434
  iscntrl, 434
  isdigit, 434
  isgraph, 434
  islower, 21, 434
  isprint, 21, 434
  ispunct, 21, 434
  isspace, 21, 434
  isupper, 21
  isxdigit, 21, 434
  toascii, 21, 796
  tolower, _tolower, 21, 796
  toupper, _toupper, 21, 796
Characters
  converting. *See* Character classification and conversion functions
  device, 437
  reading
    fgetc, fgetchar, 278
    from console, 348
    from port, 425
    getc, getchar, 346
    read function, 605
  ungetting, 805, 807
  writing
    fputc, fputchar, 305
    putc, putchar, 589
    to console, 591
    to port, 532
    write function, 826
chdir, 23, 145
_chdrive, 147
Child process
  cwait
    signal settings, 707
    termination-status word, 177, 707, 820
  exec, 251
  floating-point state of parent, 300
  spawn, 707
  wait, 820
chmod, 24, 149
chsize, 24, 151
_clear87, 46, 153
clearerr, 37, 155
_clearscreen, 29, 157
Clipping regions, 650
clock, 60, 159
clock_t type, 68
close, 42, 161
Comparison
  max macro, 484
  min macro, 506
Compatibility mode, 704
complex type, 68
CONIO.H, 44
Console, ungetting characters from, 807
_control87, 46, 163

Conversion
characters. *See* Character classification and conversion functions
data. *See* Data conversion
floating-point numbers
IEEE double to MS binary double, 181
to integers and fractions, 513
to strings, 243, 267, 340
integers to strings, 438
long integers to strings, 471, 801
strings to
floating-point values, 98
lowercase, 750
uppercase, 780
time. *See* Time, conversion
cos, cosh, 46, 166
Cosine, 166
cosl, coshl, 46
cprintf, 8, 44, 168
cputs, 44, 170
creat, 42, 171
cscanf, 8, 44, 173
ctime, 60, 175
CTYPE.H routines, 22, 434
cwait, 177

D

Data conversion
See also Conversion
atof, atoi, atol, _atold, 22, 98
ecvt, 22, 243
fcvt, 22, 267
gcvt, 22, 340
include files, 22
itoa, 22, 438
ltoa, 22, 471
strtod, strtol, strtoul, 22, 775
ultoa, 23, 801
Data items
reading, 308
writing, 338
Date routines. *See* Time, routines
Daylight variable, 64, 799
Default translation mode
child process, used in, 707
_fmode, 66
_fopen, 296
_fsopen, 327
O_TEXT, 523
setmode, 664
sopen, 704
dieeetomsbin, dmsbintoieee, 46, 181
difftime, 61, 182
DIRECT.H, 23
Directories
creating, 507
deleting, 624
getting current, 354, 356
renaming, 620
Directory control
chdir, 23
chmod, 149
getcwd, 23, 354
_getdcwd, 356
include files, 23
mkdir, 23, 507
remove, 619
rmdir, 23
unlink, 809
_disable, 57, 60, 184
diskfree_t structure, 69
diskinfo_t structure, 69
_displaycursor, 185
div, 187
div_t type, 69
Division
div, 187
ldiv, 445
Document conventions, ix
DOMAIN, 480
DOS commands, execution within programs, 784
DOS error codes, 65
DOS interface routines
bdos, 57, 101
_bios_disk, 115
_bios_equiplist, 119
_bios_keybrd, 121
_bios_memsize, 124
_bios_printer, 125
_bios_timeofday, 130
_chain_intr, 143
_disable, 184
_dos_allocmem, 189
_dos_close, 191
_dos_creat, _dos_creatnew, 193
_dos_findnext, 195
_dos_freemem, 198
_dos_getdate, 200
_dos_getdiskfree, 202
_dos_getdrive, 204
_dos_getfileattr, 206
_dos_getftime, 208
_dos_gettime, 211

DOS interface routines (*continued*)
_dos_getvect, 213
_dos_keep, 214
_dos_open, 216
_dos_read, 219
_dos_setblock, 221
_dos_setdate, 223
_dos_setdrive, 225
_dos_setfileattr, 227
_dos_setftime, 229
_dos_settime, 232
_dos_setvect, 234
_dos_write, 237
dosexterr, 239
_enable, 59, 247
FP_OFF, 59
harderr, _hardresume, _hardretn, 59
include files, 57
int86, 59, 426
int86x, 428
intdos, 59, 430
intdosx, 432
segread, 59, 640
and uses (list), 57
DOS interrupts, invoking, 426, 428
DOS system calls
_bios_serialcom, 127
error handling, 239
invoking, 101, 430, 432
DOS version number, detection, 67
DOS.H, 57
_dos_allocmem, 57, 189
_dos_close, 57, 191
_dos_creat, 57, 193
_dos_creatnew, 58, 193
dosdate_t structure, dostime_t structure, 69
_doserrnovariable, 65
DOSERROR type, 69, 239
dosexterr, 59, 239
_dos_findfirst, 58, 195
_dos_findnext, 58, 195
_dos_freemem, 58, 198
_dos_getdate, 58, 200
_dos_getdiskfree, 58, 202
_dos_getdrive, 58, 204
_dos_getfileattr, 58, 206
_dos_getftime, 58, 208
_dos_gettime, 58, 211
_dos_getvect, 58, 213
_dos_keep, 58, 214
_dos_open, 58, 216
_dos_read, 58, 219
_dos_setblock, 58, 221
_dos_setdate, 58, 223
_dos_setdrive, 58, 225
_dos_setfileattr, 58, 227
_dos_setftime, 58, 229
_dos_settime, 59, 232
_dos_setvect, 59, 234
_dos_write, 59, 237
Drive routines
_chdrive, 147
_getdrive, 359
dup, dup2, 42, 241
Dynamic allocation. *See* Memory allocation

E

E2BIG, 66
EACCES, 66
EBADF, 66, 826
ecvt, 22, 243
EDEADLOCK, 66
EDOM, 66
EEXIST, 66
EINVAL, 66
_ellipse, _ellipse_w, _ellipse_wxy, 30, 245
Ellipses, x
EMFILE, 66
_enable, 59, 247
End-of-file
indicators, 155
low-level I/O, 249
stream I/O
clearing, 155, 622
described, 272
_endthread, 248
ENOENT, 66
ENOEXEC, 66
ENOMEM, 66
ENOSPC, 66, 826
environ variable, 67–68, 360, 592
Environment variables
described, 68
getenv, 360
putenv, 592
eof, 42, 249
EOF constant, 37
ERANGE, 66
errno variable
and perror, strerror, 13
described, 65
error numbers, 538, 742

errno variable (*continued*)
graphics, routines, 13
I/O routines, 13, 43
math routines, 13
Error handling
DOS error codes, 65
DOS system calls, 239
logic errors, 92
perror, 13, 538
strerror, _strerror, 13, 742
Error indicator
described, 41, 155
ferror, 274
return value, 13
Error messages, user supplied, 538, 742
Euclidean distance, 421
exception type, 69, 480
EXDEV, 66
exec family, 8, 52, 251
exit, _exit, 52, 256
Exiting processes, 256
exp, 46, 258
_expand, 48, 260
expl, 46, 258
Exponential functions
exp, 258
expl, 258
frexp, 318
frexpl, 318
ldexp, 443
ldexpl, 443
log, log10, 459
logl, log10l, 459
pow, 578
powl, 578
sqrt, 717
sqrtl, 717

F

fabs, 46, 263
fabsl, 46, 263
Far pointers, 298
_fcalloc, 136
fclose, fcloseall, 37, 265
fcvt, 22, 267
fdopen, 37, 269
feof, 37, 272
ferror, 37, 274
_fexpand, 260
fflush, 37, 276
_ffree, 48, 310
fgetc, fgetchar, 37, 278
fgetpos, 37, 280
fgets, 37, 282
_fheapchk, 48, 409
_fheapmin, 411
_fheapset, 49, 412
_fheapwalk, 49, 415
fieeetomsbin, fmsbintoieee, 47
FILE
pointer, 37
structure, 37
type, 69
File handles
duplication, 241
functions, 42
predefined, 43
stream, 287
File handling
access, 24, 80
chmod, 24
chsize, 24, 151
filelength, 24, 285
fstat, 24, 329
include files, 23
isatty, 23, 437
locking, 23, 456
mktemp, 23, 509
remove, 23
rename, 24, 620
setmode, 24, 664
stat, 24, 723
umask, 24, 803
unlink, 24
File permission mask. *See* Permission setting
File pointers
defined, 41
positioning
fgetpos, 280
fseek, 322
fsetpos, 324
ftell, 332
lseek, 468
read and write operations, 43
rewind, 622
tell, 788
File status information, 329, 723
filelength, 24, 285
fileno, 37, 287
Files
changing size, 151
closing, 43, 161
creating, 171, 523, 704

Files (*continued*)
deleting, 619, 809
determining length, 285
locking, 456
modifying names, 509
names, 8
obtaining status, 329, 723
opening
creat, 171
input and ouput, 42
open, 523
sopen, 704
reading characters, 605
renaming, 620
setting modification time, 811
writing characters, 826
find_t structure, 69
Floating point
control word, 163
errors, 300
math package
_clear87, 153
_control87, 163
_fpreset, 300
reinitialization, 300
_status87, 725
numbers, conversion to strings, 243, 267, 340
routines, 15
status word, 153, 725
_floodfill,_floodfill_w, 30, 288
floor, 47, 290
floorl, 47, 290
flushall, 37, 292
Flushing buffers, 276, 292
_fmalloc, 49, 476
_fmemccpy, 20, 487
_fmemchr, 20, 489
_fmemcmp, 491
_fmemcpy, 20, 494
_fmemicmp, 20, 497
_fmemmove, 20, 501
_fmemset, 20, 504
fmod, 47, 293
_fmode variable, 66
fmodl, 47, 293
_fmsize, 49, 519
Fonts
bit-mapped, 656
functions (list), 32
fopen, 37, 295
Formatted I/O
cprintf, 168
cscanf, 173
fprintf, 303
fscanf, 320
printf, 580
scanf, 630
sprintf, 715
sscanf, 720
vfprintf, vprintf, vsprintf, 817
FP_OFF, FP_SEG, 59, 298
fpos_t type, 69
_fpreset, 47, 300
fprintf, 8, 38, 303
fputc, fputchar, 38, 305
fputs, 38, 307
fread, 38, 308
_frealloc, 607
free, 48, 310
_freect, 48, 313
freopen, 38, 315
frexp, 47, 318
frexpl, 47, 318
fscanf, 8, 38, 320
fseek, 38, 322
fsetpos, 38, 324
_fsopen, 38, 326
fstat, 24, 329
_fstrcat, 727
_fstrchr, 729
_fstrcmp, 731
_fstrcpy, 734
_fstrcspn, 736
_fstrdup, 740
_fstricmp, 746
_fstrlen, 748
_fstrlwr, 750
_fstrncat, 752
_fstrncmp, 754
_fstrncpy, 756
_fstrnicmp, 758
_fstrnset, 759
_fstrpbrk, 761
_fstrrchr, 763
_fstrrev, 765
_fstrset, 767
_fstrspn, 769
_fstrstr, 771

_fstrtok, 778
_fstrupr, 780
ftell, 38, 332
ftime, 61, 334
_fullpath, 336
Functions
declarations, 7–9
vs. macros, 10–12
fwrite, 38, 338

G

gcvt, 22, 340
_getactivepage, 342
_getarcinfo, 344
_getbkcolor, 29, 345
getc, getchar, 38, 346
getch, getche, 44, 348
_getcolor, 29, 350
_getcurrentposition, _getcurrentposition_w, 30, 352
getcwd, 23, 354
_getdcwd, 356
_getdrive, 359
getenv, 360
_getfillmask, 29, 362
_getfontinfo, 364
_getgtextextent, 365
_getgtextvector, 366
_getimage, _getimage_w, _getimage_wxy, 32, 367
_getlinestyle, 29, 370
_getphyscoord, 27, 372
getpid, 52, 373
_getpixel, _getpixel_w, 30, 374
gets, 38, 376
_gettextcolor, 31, 377
_gettextcursor, 378
_gettextposition, 31, 379
_gettextwindow, 381
_getvideoconfig, 27, 382
_getviewcoord, _getviewcoord_w,
_getviewcoord_wxy, 27, 386
_getvisualpage, 388
getw, 38, 389
_getwindowcoord, 26, 391
_getwritemode, 392
Global variables
accessing, 63
_amblksiz, 63
daylight, 64, 799
_doserrno, 65
environ, 67, 360, 592
Global variables (*continued*)
errno
described, 65
perror, 538
strerror, 742
_fmode, 66
_osmajor, 67
_osminor, 67
_psp, 68
sys_errlist
described, 65
perror, 538
strerror, 742
sys_nerr, 65, 538, 742
timezone, 64, 799
tzname, 64, 799
gmtime, 61, 394
Goto, nonlocal, 463, 660
Graphics
attributes, 29
color selection, 29, 647
configuration, 26, 680, 690
coordinates, 26, 650, 686, 688
font functions (list), 32
image transfer, 31
low-level palettes, 28
output, 245, 449, 517, 610
parameters, 652, 654, 661
presentation graphics, 33–34, 540, 544, 546, 550
text output, 30
text support
_gettextwindow, 381
_scrolltextwindow, 635
_settextrows, 675
_settextwindow, 677
_setvideomoderows, 684
_setwindow, 691
_wrapon, 824
Greenwich mean time, 394
_grstatus, 396

H

halloc, 48, 400
Handle. *See* File handles
_harderr, 59
_hardresume, 59
_hardretn, 59
Header files. *See* Include files
Heap consistency check
_bheapchk, 409
_bheapmin, 411

Heap consistency check (*continued*)
  _fheapchk, _heapchk, _nheapchk, 409
  _fheapmin, _heapmin, _nheapmin, 411
_heapadd, 406
_heapchk, 409
_heapmin, 411
_heapset, 412
_heapwalk, 415
hfree, 49, 419
_huge data items, 16–17
Hyperbolic
  cosine, 166
  sine, 702
  tangent, 786
hypot, 47, 421
Hypotenuse, 421
hypotl, 47, 421

I

IEEE format, converting double-precision to Microsoft
  binary, 181
_imagesize, _imagesize_w, _imagesize_wxy, 32, 423
#include directive, 6
Include files
  buffer manipulation routines, 20
  character classification, conversion, 22
  console and port I/O, 44
  Contents, 5, 7
  data conversion, 22
  directory control, 23
  DOS interface routines, 57
  file handling, 23
  low-level I/O, 42
  math routines, 46
  memory allocation, 48
  naming conventions, vi
  process control, 51
  processor calls, 61
  reasons for using, 6
  searching and sorting, 55
  stream I/O, 36
  string manipulation, 55
  time routines, 61
inp, inpw, 44–45, 425
int86, 59, 426
int86x, 59, 428
intdos, 59, 430
intdosx, 59, 432
Integers
  conversion to strings, 438
  long, conversion to strings, 471, 801
Interrupt signals, 696
Interrupts. *See* DOS interrupts, invoking
I/O
  See also Formatted I/O
  buffered, 37
  console and port
    cgets, 44, 141
    cprintf, 44, 168
    cputs, 170
    cscanf, 44, 173
    described, 35
    getch, getche, 44, 348
    include files, 44
    inp, inpw, 44, 425
    kbhit, 44, 440
    outp, outpw, 44, 532
    putch, 44, 591
    ungetch, 44, 807
  low-level
    close, 42, 161
    creat, 42, 171
    described, 35
    dup, dup2, 42, 241
    eof, 42, 249
    error handling, 43
    include files, 42
    lseek, 42, 468
    open, 42, 523
    read, 42, 605
    sopen, 42, 704
    tell, 42, 788
    write, 42, 826
  stream, 35–36
IO.H, 23, 42
isalnum, isdigit, isgraph, 21, 434
isalpha, isascii, iscntrl, 434
isatty, 24, 437
isdigit, 434
islower, isupper, isxdigit, 21, 434
isprint, 21, 434
ispunct, 21, 434
isspace, 21, 434
Italic letters, use of, ix
itoa, 22, 438

J

j0, j1, jn, 107
_j0l, _j1l, _jnl, 107
jmp_buf type, 69

K

kbhit, 44, 440
Keystroke, testing, 440

L

labs, 441
ldexpl, 47, 443
ldiv, 445
ldiv_t type, 69
lfind, 55, 447
Library (.LIB) files
  contents, 5
  default, 6
  GRAPHICS.LIB, 6
  use, 6
Library routines, calling, 5
Lines
  reading, 282, 376
  writing, 596
_lineto, 30, 449
_lineto_w, 30, 449
Local time corrections, 64, 454, 799
localeconv, 451
Localization
  localeconv, 451
  setlocale, 662
localtime, 61, 454
locking, 24, 456
log, log10, 459
Logarithmic functions, 47, 459
logl, log10l, 459
long double functions, 461
Long integers, conversion to strings, 471
longjmp, 463
Low-level graphics
  See also individual function names
  color selection, 28–29
  configuration, 26
  coordinates, 26
  font functions. *See* Fonts
  image transfer, 31
  output
    _arc, _arc_w, _arc_wxy, 30, 86
    _ellipse, _ellipse_w, _ellipse_wxy, 30, 245
    _getarcinfo, 344
    _getwritemode, 392
    _grstatus, 396
    _lineto, _lineto_w, 30, 449
    _pie, _pie_w, _pie_wxy, 30, 567
    _polygon, _polygon_w, _polygon_wxy, 574
Low-level graphics (*continued*)
  output (*continued*)
    _rectangle, _rectangle_w, _rectangle_wxy, 30, 610
    _setwritemode, 695
  palettes, 28
  parameters, 30
  physical coordinates, 26
  text support (list), 30
  view coordinates, 26
  window coordinates, 26
_lrotl, 465
_lrotr, 465
lsearch, 55, 466
lseek, 42, 468
ltoa, 22, 471

M

_makepath, 473
Macros, 10–12
malloc, 49, 476
MALLOC.H, 48
Mask. *See* Permission setting
MATH.H, 22, 46
matherr
  described, 480
  use, 47
_matherrl, 480
max, 484
_memavl, 49, 485
memccpy, 20, 487
memchr, 20, 489
memcmp, 20, 491
memcpy, 20, 494
memicmp, 20, 497
_memmax, 49, 499
memmove, 20, 501
Memory allocation
  _amblksiz, 63
  available memory, determination, 313
  _bcalloc, 136
  _bfree, 310
  _bfreeseg, 110
  _bheapadd, 406
  _bheapchk, 409
  _bheapmin, 411
  _bheapseg, 112
  _bheapset, 412
  _bheapwalk, 415
  _bmalloc, 476
  _bmsize, 519

Memory allocation (*continued*)
- _brealloc, 607
- calloc, 136
- _expand, 260
- _fcalloc, 136
- _ffree, 310
- _fheapchk, 409
- _fheapmin, 411
- _fheapset, 412
- _fheapwalk, 415
- _fmalloc, 476
- _fmsize, 519
- _frealloc, 607
- free, 310
- _freect, 313
- halloc, 400
- _heapadd, 406
- _heapchk, 409
- _heapmin, 411
- _heapset, 412
- _heapwalk, 415
- hfree, 419
- malloc, 476
- _memavl, 485
- _memmax, 499
- _msize, 519
- _ncalloc, 136
- _nfree, 310
- _nheapchk, 409
- _nheapmin, 411
- _nheapset, 412
- _nheapwalk, 415
- _nmalloc, 476
- _nmsize, 519
- _nrealloc, 607
- realloc, 607
- routines and uses (list), 48
- stackavail, 722

MEMORY.H, 20
memset, 20, 504
Microsoft Windows, 25
min, 506
mkdir, 23, 507
mktemp, 24, 509
mktime, 61, 511
Model-independent memory routines, 20
modf, 47, 513
modfl, 47, 513
Modification time, 811
Monofont, use of, x
movedata, 515
_moveto, 30, 517
_moveto_w, 30, 517
MS C 4.0, differences, puts, 596
_msize, 49, 519

N

_ncalloc, 136
NDEBUG, 92
_nexpand, 260
_nfree, 48, 310
_nheapchk, 48, 409
_nheapmin, 411
_nheapset, 49, 412
_nheapwalk, 49, 415
_nmalloc, 49, 476
_nmsize, 49, 519
Nonlocal goto, 463, 660
_nrealloc, 607
_nstrdup, 740
Null pointer, 37

O

Object (.OBJ) files, 6
O_BINARY, 66
oflag. *See* Open flag
onexit, 52, 521
open, 8, 42, 523
Open flag, 523, 704
Operating system, 14
Optional items, ix
_osmajor variable, 67
_osminor variable, 67
_outgtext, 32, 527
_outmem, 530
outp, outpw, 44–45, 532
Output. *See* I/O
_outtext, 31, 535
OVERFLOW, 480
Overlapping moves, 494
Overlay, of parent process, 707

P

Palettes, low-level, 28
Parameters, variable-length number, 62, 817
Parent process
- cwait, 177
- described, 251
- overlay and suspension, 707
- wait, 820

Path names, 9

_pclose, 537
Permission setting
  access, 80
  changing, 149
  described, 171
  mask, 803
  open, 523
  sopen, 704
  umask, 803
perror, 13, 538
_pg_analyzechart, 34, 540
_pg_analyzechartms, 34, 540
_pg_analyzepie, 34, 543
_pg_analyzescatter, 34, 544
_pg_analyzescatterms, 34, 544
_pg_chart, 33, 546
_pg_chartms, 33, 546
_pg_chartpie, 33, 549
_pg_chartscatter, 33, 550
_pg_chartscatterms, 33, 550
_pg_defaultchart, 33, 552
_pg_getchardef, 34, 554
_pg_getpalette, 34, 555
_pg_getstyleset, 34, 558
_pg_hlabelchart, 34, 559
_pg_initchart, 33, 560
_pg_resetpalette, 34, 561
_pg_resetstyleset, 34, 562
_pg_setchardef, 34, 563
_pg_setpalette, 34, 564
_pg_setstyleset, 34, 565
_pg_vlabelchart, 34, 566
_pie, _pie_w, _pie_wxy, 30, 567
_pipe, 570
Pipes
  _pclose, 537
  _pipe, 570
  _popen, 576
PLOSS, 480
Pointers, long, 298
_polygon, _polygon_w, _polygon_wxy, 574
_popen, 576
Port I/O. *See* I/O, console and port
pow, 47, 578
Predefined
  handles, 43
  stream pointers, 39
  types. *See* Standard types
printf, 38, 580
Printing. *See* Write operations
Process control
  abort, 51, 76
  atexit, 51, 96
  _cexit, _c_exit, 140
  cwait, 177
  exec family, 52
  exit, _exit, 52, 256
  getpid, 52, 373
  include files, 51
  onexit, 52, 521
  raise, 52, 601
  signal, 52, 696
  spawn family, 53
  system, 53, 784
  wait, 820
Process ID, 373
PROCESS.H, 51
Processor calls, include files, 61
Program segment prefix (PSP), 68
Pseudorandom integers, 603, 718
_psp, 68
putc, putchar, 38, 589
putch, 44, 591
putenv, 592
_putimage, _putimage_w, 32, 594
puts, 38
putw, 38, 597

Q

qsort, 55, 599
Quick sort algorithm, 599
Quotation marks, use of, x

R

raise, 52, 601
rand, 603
Random access
  fgetpos, 280
  fseek, 322
  fsetpos, 324
  ftell, 332
  lseek, 468
  rewind, 622
  tell, 788
Random number generator, 603, 718
read, 42, 605
Read access. *See* Permission setting

Read operations
binary int value, 389
characters
from file, 605
from stdin, 278, 346
from stream, 278, 346
from console, 141, 173, 348, 440
formatted
cscanf, 173
fscanf, 320
scanf, 630
sscanf, 720
line
from stdin, 376
from stream, 282
from port, 425
realloc, 49, 607
Reallocation
_brealloc, 607
_expand, 260
_frealloc, 607
_nrealloc, 607
realloc, 49, 607
_rectangle, _rectangle_w, _rectangle_wxy, 30, 610
Redirection, 40, 43–44, 315
_registerfonts, 32, 612
REGS type, 70
Remainder function, 293
_remapallpalette, 28, 613
_remappalette, 28, 613
remove, 24, 619
rename, 24, 620
Reversing strings, 765
rewind, 38, 622
rmdir, 23, 624
rmtmp, 38, 626
_rotl, 628
_rotr, 628

S

scanf, 8, 38, 630
Scanning. *See* Read operations
_scrolltextwindow, 635
SEARCH.H, 55
_searchenv, 638
Searching and sorting
bsearch, 55, 132
include files, 55
lfind, 447
lfind, lsearch, 55
Searching and sorting (*continued*)
lsearch, 466
qsort, 55, 599
seed, 718
Segment registers, 640
segread, 59, 640
_selectpalette, 28, 642
_setactivepage, 26, 645
_setbkcolor, 29, 647
setbuf, 38, 40, 648
_setcliprgn, 650
_setcolor, 29, 652
_setfillmask, 29, 654
_setfont, 32, 656
_setgtextvector, 659
setjmp, 660
_setlinestyle, 29, 661
setlocale, 662
setmode, 24, 664
_setpixel, _setpixel_w, 29, 666
_settextcolor, 31, 668
_settextcursor, 671
_settextposition, 673
_settextrows, 675
_settextwindow, 31, 677
setvbuf, 38, 40
_setvideomode, 26, 680
_setvideomoderows, 684
_setvieworg, 27, 686
_setviewport, 27, 688
_setvisualpage, 26, 690
_setwindow, 27, 691
_setwritemode, 695
signal
described, 53, 696
raise, 601
SIGNAL.H, 51
sin, sinh, 47, 702
Sine, 702
SING, 480
sinl, sinhl, 47, 702
size_t type, 70
Small capital letters, use of, x
sopen, 8, 42, 704
Sorting. *See* Searching and sorting
spawn family
argument-type-checking limitations, 8, 707
described, 707
use, 53
_splitpath, 713
sprintf, 8, 715
sqrt, 47, 717

sqrtl, 47, 717
Square-root function, 717
srand, 718
SREGS type, 70
sscanf
 described, 720
 type checking, 8
 use, 38
Stack checking, 12
Stack environment
 restoring, 463
 saving, 660
stackavail, 49, 722
Standard auxiliary. *See* stdaux, stderr, stdin
Standard error. *See* stdaux, stderr, stdin
Standard input. *See* stdaux, stderr, stdin
Standard output. *See* stdout, stdprn
Standard print. *See* stdout, stdprn
Standard types
 clock_t, 68
 complex, 68
 diskfree_t, 69
 diskinfo_t, 69
 div_t, 69
 dosdate_t, 69
 DOSERROR, 69, 239
 dostime_t, 69
 exception, 69, 480
 FILE, 69
 find_t, 69
 fpos_t, 69
 jmp_buf, 69
 ldiv_t, 69
 listed, 68
 REGS, 70
 size_t, 70
 SREGS, 70
 stat, 723
 time_t, 70, 182
 timeb, 70, 334
 tm, 70, 394
 utimbuf, 70, 811
 va_list, 70
stat routine
 described, 723
 use, 24
stat type
 described, 70
 fstat, 329
_status87, 725
stdaux, stderr, stdin
 buffering, 40
 described, 40
 file handle, 43
 translation mode, changing, 664
STDIO.H, 36
stdout, stdprn
 buffering, 40
 described, 40
 file handle, 43
 translation mode, changing, 664
strcat, 55, 727
strchr, 55, 729
strcmp, 55, 731
strcoll, 733
strcpy, 55, 734
strcspn, 55, 736
_strdate, 61, 738
strdup, 55, 740
Stream I/O
 See also I/O, console and port
 buffering, 40
 clearerr, 155
 described, 35
 error handling, 41
 fclose, fcloseall, 265
 fdopen, 269
 feof, 272
 ferror, 274
 fflush, 276
 fgetc, fgetchar, 278
 fgetpos, 280
 fgets, 282
 fileno, 287
 flushall, 292
 fopen, 295
 fprintf, 303
 fputc, fputchar, 305
 fputs, 307, 596
 fread, 308
 freopen, 315
 fscanf, 320
 fseek, 322
 fsetpos, 324
 _fsopen, 326
 ftell, 332
 fwrite, 338
 getc, getchar, 346
 gets, 376
 getw, 389
 printf, 580
 putc, putchar, 589

Stream I/O (*continued*)
putw, 597
rewind, 622
routines and uses (list), 37
scanf, 630
setbuf, 648
sprintf, 715
sscanf, 720
tempnam, tmpnam, 38
ungetc, 805
vfprintf, vprintf, vsprintf, 817
Stream pointer, 37
Streams
appending, 295, 315, 326
buffering, 648
clearing errors, 155
closing, 41, 265
file handles for, 287
file pointer position
fseek, 322
fsetpos, 324
ftell, 332
fgetpos, 280
rewind, 622
formatted I/O
printf, 580
scanf, 630
sprintf, 715
sscanf, 720
stream, 303, 320
vprintf, 817
opening, 39, 269, 295, 326
reading
binary int value, 389
characters, 278, 346
data items, 308
lines, 282, 376
reopening, 315
rewinding, 622
stdaux, stderr, stdin, 40
stdout, stdprn, 40
translation mode. *See* Binary, mode
ungetting characters, 805
writing
binary int value, 597
characters, 305, 589
data items, 338
lines, 596
strings, 307
strerror, 13, 55, 742
_strerror, 742
strftime, 744
stricmp, 55, 746
String manipulation
_fstrcat, 727
_fstrchr, 729
_fstrcmp, 731
_fstrcpy, 734
_fstrcspn, 736
_fstrdup, 740
_fstricmp, 746
_fstrlwr, 750
_fstrncat, 752
_fstrncmp, 754
_fstrncpy, 756
_fstrnicmp, 758
_fstrnset, 759
_fstrpbrk, 761
_fstrrchr, 763
_fstrrev, 765
_fstrset, 767
_fstrspn, 769
_fstrstr, 771
_fstrtok, 778
_fstrupr, 780
_nstrdup, 740
routines and uses (list), 55
strcat, 727
strchr, 729
strcmp, 731
strcoll, 733, 744
strcpy, 734
strcspn, 736
strdup, 740
stricmp, 746
strlwr, 750
strncat, 752
strncmp, 754
strncpy, 756
strnicmp, 758
strnset, 759
strpbrk, 761
strrchr, 763
strrev, 765
strset, 767
strspn, 769
strstr, 771
strtok, 778
strupr, 780
strxfrm, 782
STRING.H, 55

Strings
  comparing, 731, 733, 744, 746, 754, 758
  concatenating, 752
  converting
    to floating-point values, 98
    to lowercase, 750
    to uppercase, 780
  copying, 734, 740, 756
  initializing, 759, 767
  reading from console, 141
  reversing, 765
  searching
    _fstrchr, 729
    _fstrcspn, 736
    _fstrpbrk, 761
    _fstrrchr, 763
    _fstrspn, 769
    _fstrstr, 771
    _fstrtok, 778
    strchr, 729
    strcspn, 736
    strpbrk, 761
    strrchr, 763
    strspn, 769
    strstr, 771
    strtok, 778
  writing
    to console, 168, 170
    to stream, 307
strlen, 56
strlwr, 56, 750
strncat, 56, 752
strncmp, 56, 754
strncpy, 56, 756
strnicmp, 56, 758
strnset, 56, 759
strpbrk, 56, 761
strrchr, 56, 763
strrev, 56, 765
strset, 56, 767
strspn, 56, 769
strstr, 56, 771
_strtime, 61, 773
strtod, 22, 775
strtok, 56, 778
strtol, 23, 775
_strtold, 23, 775
strtoul, 23, 775
strupr, 56, 780
strxfrm, 782
swab, 783
SYS\STAT.H, 23
SYS\TIMEB.H, 61
SYS\TYPES, 61
SYS\UTIME.H, 61
sys_errlist
  described, 65
  system error messages, 538, 742
sys_nerr, 65, 538, 742
system, 53, 784
System calls. *See* DOS system calls
System time. *See* Time

T

tan, tanh, 47, 786
Tangent, 786
tanl, tanhl, 47, 786
tell, 42, 788
tempnam, tmpnam, 38, 790
Terminal capabilities, 437
Text mode
  vs. binary, 35
  described, 66, 523
  setmode, 664
  sopen, 704
  stream I/O, 269, 296, 315, 327
Threads
  _beginthread, 103
  DosExit, 248
  _endthread, 248
  termination, 248
Time
  conversion
    long integer to string, 175
    long integer to structure, 454
    structure to string, 88
  functions, 61
  global variables, setting, 799
  local time, correcting, 454
  obtaining, 334, 793
  routines
    asctime, 88
    clock, 159
    ctime, 175
    difftime, 182
    ftime, 334
    gmtime, 394
    (list), 60
    localtime, 454
    mktime, 511
    time, 793
    tzset, 799
    utime, 811

Time (*continued*)
    time differences, computing, 182
time function, 793
TIME.H, 61
time_t type, 70, 182
timeb type, 70, 334
timezone variable, 64, 799
TLOSS, 480
tm type, 70, 394
tmpfile, 39
tmpnam, 790
toascii, 21, 796
Tokens, finding in strings, 778
_tolower, _toupper, 21, 796
tolower, toupper, 21, 796
_toupper, 796
Trigonometric functions
    acos, 82
    acosl, 82
    asin, 90
    asinl, 90
    atan, atan2, 94
    atanl, atan2l, 94
    cos, cosh, 166
    hypot, 421
    hypotl, 421
    sin, sinh, 702
    sinl, sinhl, 702
    tan, tanh, 786
    tanl, tanhl, 786
Type checking
    function declarations, 7–8
    include files, 7
    variable arguments, 8
TZ environment variable
    default value, 64
    localtime, 454
    tzset, 799
tzname variable, 64, 799
tzset, 799

U

ultoa, 23, 801
umask, 24, 803
UNDERFLOW, 481
UNIX, 9
ungetc, 39, 805
ungetch, 44, 807
Universal Coordinated Time, 61
unlink, 24, 809
_unregisterfonts, 32, 810
Uppercase, use of, ix
utimbuf type, 70, 811
utime, 811

V

va_arg, va_end, va_start, 62
va_list type, 70
Version number (DOS), 67
vfprintf, vprintf, vsprintf, 39, 817

W

wait, 820
Word. *See* Binary, int
_wrapon, 31, 824
write, 42, 826
Write access. *See* Permission setting
Write operations
    binary int value to stream, 597
    character
        to console, 168
        to file, 826
        to stdout, 305, 589
        to stream, 305, 589, 805
    to console, 169, 170, 591
    data items from stream, 338
    formatted
        cprintf, 168
        printf, 580
        sprintf, 715
        stream I/O, 303
        vprintf, 817
    line to stream, 596
    to port, 532
    string to stream, 307

X

XENIX, 9

Y

y0, y1, yn, 107
_y0l, _y1l, _ynl, 107

OTHER TITLES FROM MICROSOFT PRESS

MICROSOFT® C RUN-TIME LIBRARY: PROGRAMMER'S QUICK REFERENCE
Kris Jamsa

This handy reference provides instant access to concise information on more than 250 commonly used functions and macros in the Run-Time Library for Microsoft C and Microsoft QuickC. Each entry includes complete syntax, a brief description, details on parameters, notes and comments, and usually a working example.
272 pages, softcover 4¾ x 8 $7.95 ISBN 1-55615-227-2

ADVANCED MS-DOS® PROGRAMMING, 2nd ed.
The Microsoft® Guide for Assembly Language and C Programmers
Ray Duncan

"ADVANCED MS-DOS exemplifies how a highly technical book can be both informative and readable." BYTE

The preeminent source of MS-DOS information for assembly language and C programmers—now completely updated with data and programming advice covering ROM BIOS for the IBM PC, PC/AT, PS/2, and related peripherals (including disk drives, video adapters, and pointing devices); MS-DOS through version 4; "well-behaved" *vs* "hardware-dependent" applications; version 4 of the Lotus/Intel/Microsoft Expanded Memory Specification; and compatibility considerations for OS/2. Duncan, a DOS authority and noted columnist, explores key programming topics, including character devices, mass storage, memory allocation and management, and process management. In addition, a healthy assortment of updated assembly language and C listings ranges from code fragments to complete utilities. The examples, from programming samples to full-length utilities, are instructive and extremely useful. All were developed using Microsoft Macro Assembler version 5.1 and Microsoft C Compiler version 5.1. And the reference section, detailing each MS-DOS function and interrupt, is virtually a book within a book. ADVANCED MS-DOS PROGRAMMING—your key to fast, efficient, robust programs.
688 pages, softcover 7⅜ x 9¼ $24.95 ISBN 1-55615-157-8

THE MS-DOS® ENCYCLOPEDIA
Foreword by Bill Gates
Ray Duncan, General Editor

"The encyclopedia...sums up the expert opinion on everything under the DOS sun in a cohesive, well-indexed and organized form." PC Week

ADVANCED OS/2 PROGRAMMING
The Microsoft® Guide to the OS/2 Kernel for Assembly Language and C Programmers
Ray Duncan

"This indispensable guide to the inner workings of the OS/2 kernel is the most important book on the subject to date....I haven't seen a book I'd recommend more than ADVANCED OS/2 PROGRAMMING." BYTE magazine

From the OS/2 Programmer's Library, here is the most complete and accurate source of information on the features and structure of OS/2. With insight and economy, Duncan explains the OS/2 services for controlling the user interface, programming mass storage, and exploiting advanced features such as multitasking and interprocess communication. Advanced chapters discuss writing filters, device drivers, monitors, and dynamic link libraries. Program examples are provided in both assembly language and C. ADVANCED OS/2 PROGRAMMING contains a complete, example-packed reference section on the more than 250 OS/2 1.1 kernel functions, with complete information on their calling arguments, return values, and special uses. ADVANCED OS/2 PROGRAMMING will transform high-level MS-DOS programmers into capable OS/2 programmers.
800 pages, softcover 7³/₈ x 9¼ $24.95 ISBN 1-55615-045-8

PROGRAMMING THE OS/2 PRESENTATION MANAGER
The Microsoft® Guide to Writing Applications for the OS/2 Graphical Windowing Environment
Charles Petzold

This is the first full discussion of the features and operation of the OS/2 1.1 Presentation Manager—the primary application environment under OS/2. It is designed to get the OS/2 application programmer—one with a background in Windows or with strong C experience—through the Presentation Manager system of windows, messages, and function calls. Charles Petzold includes scores of valuable Presentation Manager programs and utilities written in C and covers these key topics: managing window-handling input and output; working with the keyboard, mouse, and timer; controlling child windows; using bitmaps, icons, pointers, and strings; accessing menus; using keyboard accelerators; working with dialog boxes; understanding dynamic linking; and using multithread programming techniques. Endorsed by the Microsoft Systems Software group, this book is unparalleled for its clarity, detail, and comprehensiveness and should be added to any OS/2 programmer's library.
864 pages, softcover 7³/₈ x 9¼ $29.95 ISBN 1-55615-170-5

Microsoft® C
Run-Time Library Reference

Covers version 6

The Microsoft C Run-Time Library, available with Microsoft C, is a set of more than 500 functions and macros that offer extraordinary power to the C programmer. The functions cover everything from presentation graphics and low-level system calls for the DOS programmer to multiple-thread support for the OS/2 programmer. These functions are essential when you manage input and output, allocate storage, and handle process control. They simplify scores of common programming tasks, save hours of development time, and provide ready access to operating system functions.

The MICROSOFT C RUN-TIME LIBRARY REFERENCE is an up-to-date complement to Microsoft C's online reference, the Microsoft C Advisor. It provides a superb introduction to using the run-time library, its variables, and its types. A useful section identifies functions by category so that you can quickly find a library routine even if you don't know its name.

The core of the book provides detailed information on each function in the run-time library — syntax; example programs; the include file, prototypes, arguments, and return values; cross-references to related functions; and notes on compatibility with ANSI, DOS, OS/2, UNIX,® and XENIX.®

The MICROSOFT C RUN-TIME LIBRARY REFERENCE is an essential reference to the industry-standard C library.

ISBN 1-55615-225-6

9 781556 152252 90000

Programming/C

U.S.A.	**$22.95**
U.K.	£19.95
Austral.	$34.95

(recommended)