IBM C/2™ Computer Language Series

Language Reference

Programming Family

84X1794

IBM C/2™ Computer Language Series

Language Reference

Programming Family

First Edition (September 1987)

Preface

This book is volume 3 of the three – volume set explaining IBM C/2.™ It contains descriptions of each of the functions of the C run-time library. The keywords and commands are listed in alphabetical order.

This product attempts to conform to the forthcoming American National Standards Institute (ANSI) standard whenever possible. IBM C/2 does make minor changes to what is considered ANSI standard C; these changes are documented in this and other IBM publications for the IBM C/2 Compiler.

First-time users of this book are expected to be computer science students who are studying C in their second or third year of school. Experienced users are expected to be experienced applications programmers or system programmers. Users should also be familiar with their computer and operating system.

Related Publications

The following books contain topics related to information in the IBM C/2 Library:

- *IBM C/2 Fundamentals*
- *IBM C/2 Compile, Link, and Run*

- *IBM Disk Operating System User's Guide*
- *IBM Disk Operating System User's Reference*
- *IBM Disk Operating System Technical Reference*

- *IBM Operating System/2™ (OS/2™) User's Guide*
- *IBM Operating System/2 User's Reference*
- *IBM Operating System/2 Programmer's Guide*
- *IBM Operating System/2 Technical Reference*

IBM C/2 is a trademark of International Business Machines Corporation

Operating System/2 (OS/2) is a trademark of International Business Machines Corporation.

- *IBM Personal System/2™ (PS/2™) Quick Reference*
- *IBM Personal System/2 Model 50 Technical Reference*
- *IBMPersonal System/2 Model 60 Technical Reference*
- *IBMPersonal System/2 Model 80 Technical Reference*

- *IBM Personal Computer Personal Editor*
- *IBM Personal Computer Guide to Operations*
- *IBM Personal Computer Technical Reference.*

- *iAPX 86, 88 User's Manual* Copyright 1981, Intel Corp. Santa Clara, CA.
- *iAPX 286 Hardware Reference Manual* Copyright 1983, Intel Corp. Santa Clara, CA.
- *iAPX 286 Programmer's Reference Manual* Copyright 1985, Intel Corp. Santa Clara, CA.

You can use the following table as a cross reference for information in the IBM C/2 library.

If You Want To...	**Refer to...**
Install the product	Compile, Link, and Run
Learn basic facts about the language	Fundamentals
Know the syntax of an instruction	Language Reference
Understand error messages	Language Reference
Debug a program	Compile, Link, and Run
Compile a program	Compile, Link, and Run
Link a program	Compile, Link, and Run
Write a program	Fundamentals, Language Reference, and Compile, Link, and Run

---

Personal System/2 (PS/2) is a trademark of the International Business Machines Corporation.

Contents

Chapter 1. About the IBM C/2 Library

The IBM C/2 run-time library is a set of over 200 predefined functions and macros designed for use in C programs. The run-time library makes programming easier by providing:

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

The run-time library is especially important in C programming because the C language does not provide some basic functions such as input and output, storage allocation, and process control.

The math routines of the IBM C run-time library have been extended to provide exception handling.

If you are interested in taking advantage of the specific features of DOS, the library includes DOS interface functions. These functions let you make DOS system calls and interrupts from a C program. The library also contains input and output routines to allow reading from your keyboard and writing to your screen.

To take advantage of the type-checking capabilities of IBM C/2, the include files that accompany the run-time library have been expanded. In addition to the definitions and declarations required by library functions and macros, the include files now contain function declarations The argument type lists enable type-checking for calls to library routines. This feature can be extremely helpful in detecting subtle program errors resulting from type mismatches between actual and formal arguments to a function. Although using argument-type lists for type-checking is helpful, you are not required to use argument type-checking. The function declarations in the include files are in preprocessor **#ifdef** blocks; defining the LINT_ARGS identifier enables them.

To provide argument-type lists for all run-time functions, several new include files have been added to the list of standard include files.

The names of the new include files have been chosen to maintain as much compatibility as possible with the proposed ANSI standard for C.

How This Book Is Organized

Chapter 1, "About the IBM C/2 Library," tells how to use the C/2 library. It discusses functions and macros, tells how to include files, declare functions, and use the argument type-checking feature in making function calls. It tells how to use huge arrays with library functions.

Chapter 2, "Global Variables and Standard Types," is a reference to the global variables and standard types that the include files define. The global variables are in alphabetical order. The standard data types are in an alphabetical list at the end of the chapter.

Chapter 3, "Run-Time Routines by Category," is a cross-reference by category to the routines described in Chapter 5. The categories are as follows:

- Buffer manipulation
- Character classification and conversion
- Data conversion
- Directory control
- File handling
- Input and output
  - Stream routines
  - Low-level routines
  - Keyboard and port routines
- Math
- Reserving storage
- DOS interface
- Process control
- Searching and sorting
- Manipulating strings
- Time
- Variable-length argument lists
- Miscellaneous routines.

Chapter 4, "Include Files," is a cross-reference to the include files, in which the functions of the run-time library are defined.

Chapter 5, "Library Routines," is a reference to the more than 200 C functions that constitute the run-time library.

Appendix A, "Error Messages," is a comprehensive reference to the error messages that this product can display and to the error situations that produce them.

Appendix B, "Reentrant Functions," is a list of the functions that you can use as reentrant under OS/2.

Appendix C, "ASCII Characters," is a table of ASCII character codes for your IBM system unit.

The C language is a powerful, general-purpose programming language capable of producing efficient, compact, and portable code. IBM C/2 is a C language compiler that includes a large number of functions designed for application programmers.

Hardware Requirements

The following are minimum hardware requirements for IBMC/2:

- IBM Personal Computer, IBM Portable Computer, IBM Personal Computer XT, IBM Personal Computer AT, IBM Personal Computer Convertible, or IBM Personal System/2. Each system listed must have at least 320K bytes (1 K byte = 1024 bytes) of user-available memory.
- IBM Color Display with the IBM Color/Graphics Display Adapter or the IBM Monochrome Display with the IBM Monochrome Display and Printer Adapter.
- Two dual-sided diskette drives (3.5 inch only) or one dual-sided diskette (3.5 inch or 5.25 inch) drive with one fixed disk. However, using this product with a fixed disk provides optimal results.
- While a printer is optional in all cases, you should use a printer while using this product.

Software Requirements

The minimum software requirement for IBM C/2 is the IBM Disk Operating System (DOS) Version 3.30.

Notation Used In This Book

This book uses certain conventions to define operating system commands, formats of functions, names, and terms.

Typeface Notation

The following typeface conventions are used in the IBM C/2 books:

Bold: Boldface is used for:

- Anything you must type exactly as it appears in the book, such as:
  - Functions - Examples: **main**, **printf**, **open**
  - Declaratives - Example: **argc**
  - Pointers to functions
  - Keys that you press after entering a command
  - Keywords - Examples: **near**, **far**, **huge**
  - Library routines - Examples: **spawn**, **exec**, **system**.
- Anything that appears on a screen that is referred to in text.

  Example: The **Stack Overflow** message tells you....
- Single alphabetic keys on the keyboard.

  Example: Type **S** and....

Italics: Italics are used for:

- New terms when they are first defined in a book.

  Example: An *object module* is produced...
- Variables, including all-caps variables, in command formats and within text. You supply these items.

  Example: TIME [*hh.mm.ss.xx*]

- Book titles.

  Example: The *IBM C/2 Fundamentals* book....

Small Capital Letters: Small capital letters are used for:

- The names of commands

  Example: the ENTER BYTES command

- Sample filenames in text.

  Example: Use the AUTOEXEC file....

- DOS programming commands.

  Example: The COPY command....

- Suffixes (file or language extensions) used alone.

  Example: A .BAT file is required....

- All acronyms and other fully capitalized words.

  Examples: IBM, DOS

- Library names.

  Example: Place it in the LIB1.LIB....

Ellipses: Additional information that you supply in the form shown.

Brackets: Items in square brackets [ ] are optional. You must place square brackets around the subscripts of arrays.

Vertical Bars: Items separated by a vertical bar (|) mean that you can enter one of the separated items. For example:

```
ON|OFF
```

Means you can enter ON or OFF but not both.

Hexadecimal Representation

This book represents hexadecimal numbers in two ways. The letter **H** shows hexadecimal system calls, such as **59H**, in DOS.

All other hexadecimal numbers use the standard C representation **0x***hexdigits*, such as **0x1F**.

Operating Systems

Throughout these books, the references to operating systems have the following meaning:

Abbreviation	Meaning
DOS	DOS 3.30
OS/2	OS/2 Operating System

Also, throughout this book, the following terms have the specified reference:

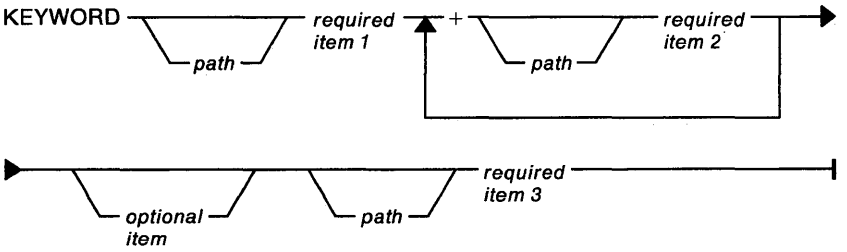
Term	Reference
Codeview®	IBM Codeview, Version 1.00
EXEMOD	IBM EXEMOD/2, Version 1.00
LIB	IBM Library Manager/2, Version 1.00
LINK	IBM Linker/2, Version 1.00
MAKE	IBM MAKE/2, Version 1.00

CodeView is a registered trademark of the Microsoft Corporation.

Command Syntax Notation

These books use syntax diagrams to explain the format of commands entered on the DOS command line. The syntax diagram has the command name at the beginning (top left corner). You follow the diagram using the reading pattern of left-to-right and top-to-bottom.

The following is a sample syntax diagram:



Syntax Diagram Terms Used

syntax diagram An illustration of possible structural patterns of a command sequence.

baseline A horizontal line that connects each of the required items in turn.

branch lines Multiple horizontal lines that show choices. Branch lines are below the base line.

keyword Words shown in all uppercase letters. Compile and utility names are keywords. You can type keywords in any combination of uppercase and lowercase letters.

variable Items shown in lowercase italic letters mean that you are to substitute the item. For example: *filename* indicates that you should type the name of your file in place of *filename*.

required items Items that must be included. Required items appear on the base line. Command names are required items.

optional items Items that you can include if you choose to do so. Optional items appear below the base line.

repeat symbol A symbol that indicates you can specify more than one choice or a single choice more than once.

Arrows are used to show base line continuation and completion as follows:

 Symbol indicates that the command syntax is continued.

 Symbol indicates that a line is continued from the previous line.

 Symbol indicates the end of a command.

 Symbol indicates that you can specify a choice more than once.

Reading Syntax Diagrams

To read a syntax diagram:

1. Start at the top left of the diagram.
2. Follow only one line at a time going from left to right and top to bottom.
3. Items on the lines indicate what you must or can specify and the required sequence.
4. When you encounter one or more branch lines, you must make a choice of items. Follow the line you choose from left to right except where you encounter the repeat symbol. The repeat symbol indicates you can make more than one choice or a single choice more than once.

With many commands, you can enter as many of a group of options as you want. These options are in a box that has a repeat arrow around it. You can follow the arrow through the box until you have selected all the options you want to use. Once you have chosen an option from the box, you cannot choose the same option again.

Using C Library Routines

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

At link time, you must link your program with the appropriate C library file or files to resolve the references to the library functions and provide the code for the called library functions. The procedures for linking with the C library are discussed in detail in the *IBM C/2 Compile, Link, and Run* book.

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

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

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

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

Identifying Functions and Macros

Most routines in the run-time library are C functions; that is, they consist of compiled C statements. However, some routines are in the form of macros. A *macro* is an identifier defined with the C preprocessor directive **#define** to represent a value or expression. Like a function, a macro can be defined to take zero or more arguments, which replace formal parameters in the macro definition. Defining and using macros is discussed in detail in the *IBM C/2 Fundamentals* book.

The macros defined in the C run-time library behave like functions. They take arguments and return values, and you call them in a similar manner. The major advantage of using macros is faster running time; the preprocessor expands their definitions, eliminating the overhead required for a function call. However, because the preprocessor expands macros (replaces them with their definitions) before compiling, using macros can increase the size of a program, particularly when a macro occurs many times in the program. Unlike a function, which is defined only once regardless of how many times it is called, each occurrence of a macro produces the expanded definition. Functions and macros thus offer a compromise between speed and size. In several cases, the C library provides both macro and function versions of the same library routine to allow you this choice.

Some important differences between functions and macros are:

- Some macros may treat arguments with side effects incorrectly when you define the macro to evaluate the arguments more than once. See the example that follows this list.
- A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address as a function identifier does. Therefore, you cannot use a macro identifier in contexts requiring a pointer. For instance, if you give a macro identifier as an argument in a function call, the program passes the *value* represented by the macro; if you give a function identifier as an argument in a function call, the program passes the *address* of the function.
- Because macros are not functions, you cannot declare them, nor can you declare pointers to macro identifiers. Thus, you cannot perform type-checking on macro arguments. The compiler does, however, detect cases in which you specified the wrong number of arguments for the macro.
- The library routines used as macros are defined through preprocessor directives in the library include files. To use a library macro, you must include the appropriate file, or the macro is undefined.

The routines used as macros are marked with a note in Chapter 5, "Library Routines." You can examine a particular macro definition in the corresponding include file to tell whether arguments with side effects can cause problems.

Example
This example uses the **toupper** routine from the C library. The **toupper** routine is a macro.

```
#include <ctype.h>

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

The include file **ctype.h** contains the following definition of **toupper**:

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

The definition uses the conditional operator (? :). In the conditional expression, the macro expansion evaluates the argument *c* twice: once to determine whether it is lowercase, and once to return the appropriate result. This causes the macro to evaluate the argument **a++** twice, increasing *a* twice instead of once. As a result, the value operated on by **islower** differs from the value operated on by **_toupper**.

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

Including Files

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

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

- **Definitions of manifest constants**. For example, the constant BUFSIZ, which determines the size of buffers for buffered input and output operations, is defined in **stdio.h**.
- **Definitions of types**. Some run-time routines take data structures as arguments or return values with structure types. Include files set up the required structure type definitions. For example, most stream input and output operations use pointers to a structure of type **FILE**, defined in **stdio.h**.

- **Two sets of function declarations**. The first set of declarations gives return types and argument type lists for run-time functions, while the second set declares only the return type. Declaring the function return type is required for any function that returns a value with type other than **int**; see "Declaring Functions" in this chapter. The presence of an argument type list enables type-checking for the arguments in a function call; see "Argument Type-Checking" in this chapter for a discussion of this option.
- **Macro definitions**. Some routines in the run-time library are macros. The definitions for these macros are in the include files. To use one of these macros, you must include the appropriate file.

The reference page for each library routine lists the include file or files needed by the routine.

Declaring Functions

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

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

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

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

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

you can also include the declaration:

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

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

You can provide your own function declarations instead of using the declarations in the library include files if you wish. However, it is recommended that you consult the declarations in the include files to make sure that your declarations are correct.

Stack Checking

Upon entry to a library routine, the routine may make a call to a stack-checking subroutine. This subroutine determines whether or not there is space on the stack for the local variables used by the routine. If enough space exists, it is allocated and the stack pointer is adjusted accordingly; otherwise a "Stack Overflow" run-time error occurs. If stack checking has been turned off, the compiler assumes there is enough stack space. If in fact the stack has too little space, you may write over memory locations in the data segment with no warning.

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

The following routines have stack checking enabled:

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

Argument Type-Checking

The IBM C/2 compiler offers a type-checking feature for the arguments in a function call. The IBM C/2 compiler checks argument types whenever an argument type list is present in a function declaration. The form of the argument-type list and the method for checking the type are discussed in full in the *IBM C/2 Fundamentals* book.

For functions that you write yourself, you must set up argument-type lists to call type-checking. You can also use the **/Zg** command line option to cause the compiler to produce a list of function declarations for all functions defined in a particular source file. You can then incorporate the list into your program. See the *IBM C/2 Compile, Link, and Run* book for details about using the **/Zg** option.

For functions in the C run-time library, you can use the procedures outlined in this section to check the type on arguments. Every function in the C run-time library is declared in one of the library include files. Two declarations are given for each function: one with and one without an argument type list. The function declarations are enclosed in an **#ifdef** preprocessor block. If you define an identifier named LINT_ARGS, the declarations containing argument type lists are processed and compiled, thus enabling argument type-checking. If the LINT_ARGS identifier is not defined, the declarations without argument type lists are included, and argument type-checking is not performed.

By default, LINT_ARGS is undefined, so no type-checking is performed for library functions. You can define LINT_ARGS in one of two ways:

- Use the **/D** command-line option to define LINT_ARGS at compile time.
- Define LINT_ARGS with a **#define** directive in your source file. The **#define** directive must occur *before* the **#include** directive for the given file to be effective.

The value of LINT_ARGS is not significant. You can define it to any value, including an empty value.

Notice that the LINT_ARGS definition applies only to the library function declarations given in the include files. The function declarations in your source program or in your own include files are not affected. You can make the inclusion of your own declarations dependent on the LINT_ARGS identifier by using an **#if** or **#ifdef** directive. Refer to the library include files for a model.

Only limited type-checking can be performed on functions that take a variable number of arguments. The following run-time functions are affected by this limitation:

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

Error Handling

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

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

When an error occurs, the C compiler sets a global variable named *errno* to a value showing the type of error. You cannot depend upon *errno* being set unless the description of the routine explicitly mentions the *errno* variable.

When using routines that set *errno*, you can test the *errno* values against the error values defined in **errno.h**, or you can use the **perror** routine to print the system error message to standard error.

For a listing of *errno* values and the associated error messages, see Appendix A, "Error Messages."

When you use *errno* and **perror**, keep in mind that the value of *errno* reflects the error value for the last call that set *errno*. To prevent misleading results, test the return value to verify that an error actually occurred before you get access to *errno*. Whenever you find that an error occurred, use *errno* or **perror** immediately. Otherwise, the value of *errno* can be changed by intervening calls.

Each math routine sets *errno* upon finding an error in the manner described on the "Library Routines" page for that routine. Math routines handle errors by calling a function named **matherr**. You can choose to handle math errors differently by writing your own error routine and naming it **matherr**. When you provide your own **matherr** function, that function is used in place of the run-time library version. To write your own **matherr** function, follow the rules on the **matherr** page in Chapter 5.

You can check for errors in stream operations by calling the **ferror** routine. The **ferror** routine detects whether the error indicator for a given stream has been set. The error indicator is automatically cleared when the stream is closed or rewound. Or you can call the **clearerr** function to reset the error indicator.

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

The **feof** routine tests for end-of-file on a given stream. You can detect an end-of-file condition in low-level input and output with the **eof** routine or when a **read** operation returns zero as the number of bytes read.

Filenames and Pathnames

Many routines in the run-time library accept strings representing pathnames and filenames as arguments. The routines process the arguments and pass them to the operating system, which is ultimately responsible for creating and maintaining files and directories. It is important to keep in mind not only the C conventions for strings, but also the operating system rules for filenames and pathnames. There are several considerations:

- case sensitivity
- Subdirectory conventions
- Delimiters for pathname components.

The C language is *case-sensitive*, meaning that it distinguishes between uppercase and lowercase letters. The DOS operating system is not case-sensitive. When getting access to files and directories on DOS, you cannot use case differences to distinguish between identical names. For example, the names "FILEA" and "fileA" are equal and refer to the same file. Storing some include files in a subdirectory named SYS is a portable convention adopted in this manual, which includes the SYS subdirectory in the specification for the appropriate include files. If you are not concerned with portability, you can disregard this convention and set up your include files accordingly. If you are concerned with portability, using the SYS subdirectory can make portability easier.

Operating systems differ in the handling of pathname delimiters. Some systems use the forward slash (/) to delimit the components of pathnames. DOS ordinarily uses the backslash (\). Within C programs, you can use either backslash or a forward slash in DOS pathnames as long as the context is unambiguous and a pathname is clearly expected.

The following routines accept string arguments that are not known in advance to be pathnames (they may be pathnames but are not required to be). In these cases, the arguments are treated as C strings, and special rules apply.

- In the **exec** and **spawn** families of routines, you pass the name of a program that is to run as a child process and then pass strings representing arguments to the child process. The pathname of the program that is to run as the child process can use either

forward slashes or backslashes as delimiters, because the compiler expects a pathname argument. You can use backslashes in any pathname arguments to the child process. The program run as the child process might expect a string argument that is not necessarily a pathname.

- In the **system** call, you pass a command to DOS; this command need not include a pathname.

In these cases, use only the backslash (\) separator as a pathname delimiter. However, in C strings, the backslash is an escape character. It signals that a special escape sequence follows. If an ordinary character follows the backslash, the compiler disregards the backslash and prints the character. Thus, to produce a single backslash in a C string, you must code the sequence ****. See the *IBM C/2 Fundamentals* book for a full discussion of escape sequences.

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

Example

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

In the example, double backslashes must be in the call to **system** to represent the pathname:

```
DIR B:\TOP\DOWN
```

Not all calls to **system** use a pathname; for example:

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

does not contain a pathname.

Binary and Text Modes

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

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

You can control the translation mode for the files used in a program 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 the letter "b" is specified in the access *type* string for the file. If you use the **open** routine, you can specify the O_BINARY flag in the *oflag* argument to open the file in binary mode. For details about these routines, see Chapter 5, "Library Routines."

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

  You can change the value of **_fmode** in one of two ways. First, you can link with the file BINMODE.OBJ (supplied with your compiler software). Linking with BINMODE.OBJ changes the initial setting of **_fmode** to O_BINARY, causing all files except **stdin**, **stdout**, and **stderr** to be opened in binary mode. This option is described in the *IBM C/2 Compile, Link, and Run* book.

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

  You can still cancel the default mode (now binary) for particular files by opening them in text mode. The **fopen** routine opens a file in text mode when the letter **t** is specified in the access *type* string for the file. If you use the **open** routine, you can specify the O_TEXT flag in the *oflag* argument to cause the file to be opened in text mode.

- The **open** routine opens the **stdin**, **stdout**, and **stderr** streams in text mode by default; it opens **stdaux** and **stdprn** in binary mode. To process **stdin**, **stdout**, or **stderr** in binary mode instead, or to process **stdaux** or **stdprn** in text mode, use the **setmode** routine.

You can also use this routine to change the mode of a file after you have opened it. The **setmode** routine takes two arguments, a file handle and a translation mode argument, and sets the mode of the file accordingly.

DOS Considerations

The version of DOS that you are using affects some routines in the run-time library. The following list describes these routines:

dosexterr, locking, sopen

These three routines are effective only on DOS Versions 3.00 and later. The **sopen** function opens a file with file-sharing attributes. Use this function in place of **open** when you want a file to have such attributes. The **locking** function locks all or part of a file from access by other users. The **dosexterr** function provides error-handling for DOS system call 59H, and is not available in OS/2. mode.

dup, dup2

In certain cases, using the **dup** and **dup2** functions on versions of DOS earlier than 3.00 might cause unexpected results. When you use **dup** or **dup2** to create a duplicate file handle for **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** under versions of DOS earlier than 3.00, calling the **close** function with either handle causes errors in later input or output operations using the other handle. Under DOS Version 3.30 , the **close** is handled correctly and does not cause later errors.

exec, spawn

When using the **exec** and **spawn** families of routines under versions of DOS earlier than 3.00, the value of the *arg0* or *argv*[0] argument is not available to you. DOS stores the character "C" in that position. Under DOS Version 3.00 or later, the value of *arg0* or *argv*[0] contains the complete command path.

To write programs that run on all versions of DOS, you can use the *_osmajor*, *_osmode*, and *_osminor* variables, discussed in Chapter 2, "Global Variables and Standard Types," to test the current operating system version number and take the appropriate action based on the result of the test.

Example

This example tests the global variable *_osmajor* to tell whether the system is to open the file TEST.DAT using the **open** routine (under versions of DOS earlier than 3.00) or the **sopen** routine (DOS Version 3.00 or later).

```
#include        <stdlib.h>

main()
{
    if (_osmajor > 2)
        printf("> 2 : _osmajor %u\n",
                        _osmajor);
    else
        printf("<= 2 : _osmajor %u\n",
                        _osmajor);
}
```

Using Floating-Point Data

The math routines supplied in the C run-time library require floating-point routines or a math coprocessor to perform calculations with real numbers. The floating-point libraries that accompany your compiler software or by a numeric coprocessor can provide this capability.

The following list is of the names of the routines that require floating-point support:

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

In addition, the **printf** routines **cprintf**, **fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, and **vsprintf** require support for floating-point input and output when you use them to print floating-point values.

---

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

The IBM C/2 compiler detects whether a program uses floating-point values; the compiler loads floating-point routines only if the program requires them. This saves considerable space for programs that do not require floating-point support.

When you use a floating-point type character in the format string for the **printf** or **scanf** functions (**cprintf**, **fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, **vsprintf**, **cscanf**, **fscanf**, **scanf**, or **sscanf**) make sure that you specify floating-point values or pointers to floating-point values in the argument list to correspond to any floating-point type characters in the format string. Floating-point arguments let the compiler detect the use of floating-point values.

For example, if you use a floating-point type character to print an integer argument, the compiler cannot detect the use of floating-point values because it does not read the format string that the **printf** and **scanf** functions use. For instance, the following program causes a run-time error:

```
main()   /* This example produces an error */
{
   long f = 10L;
   printf("%f", f);
}
```

This program produces the following message when you run it:

Floating point not loaded

The compiler does not detect the floating-point values because you gave no floating-point arguments in the call to **printf**.

The following is a correct version of the above call to **printf**.

```
printf("%f.", (double)f);
  /* CORRECT VERSION OF THE EXAMPLE */
```

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

Huge Models

When considering huge models, you can declare a huge array in one of two ways. First, in a small-, compact-, medium-, or large-storage model, you can explicitly declare an array or pointer as **huge**:

```
double huge d[100][100];
double huge *hp;
```

The first statement defines *d* as a huge array of **double** type. This type of array requires 80000 bytes of data space, which exceeds the 64K byte limit.

The second statement defines **hp** as a pointer to a huge array of **double** type. The **huge** modifier shows that any arithmetic done with this pointer must use a 32-bit address in place of the 16-bit offset arithmetic used by other pointers.

You can also declare a huge array by compiling the entire program in a huge model (**/AH** option). Because a huge model allows arrays to be larger than 64K bytes and assumes all pointers to be huge, the above examples still work, but the **huge** keyword is not required.

However, because of the 16-bit default size of small and medium models, you cannot pass huge arrays and huge pointers arrays to library functions. Huge items require a 32-bit address and therefore you cannot pass them to library functions (or any other functions) that expect a 16-bit pointer argument. When using large or huge models, you can use huge arrays and pointers as arguments to functions.

Using Huge Arrays with Library Functions

In programs that use the small-, compact-, medium-, and large-storage models, IBM C/2 lets you use arrays exceeding the 64K-byte limit of physical storage by explicitly declaring the arrays as **huge**. (See "Working with Storage Models" in the *IBM C/2 Compile, Link, and Run* book for a complete discussion of storage models and the **near**, **far**, and **huge** keywords.) However, you cannot generally pass **huge** data items as arguments to C library functions. In the case of small and medium models, where the default size of a data pointer is **near** (16 bits), the only routines that accept huge pointers are **halloc** and **hfree**. In the compact model library used by compact-model programs, and in the large model library used by both large-model and huge-model programs, only the functions listed below use argument arithmetic that works with **huge** items:

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

With this set of functions, you can read from or write to huge arrays, sort and search them, copy data from them, initialize variables in them, compare the values of elements in them, or dynamically reserve or free storage for them. You can pass any of these functions a huge pointer in a compact-, large-, or huge-model program without difficulty.

Chapter 2. Global Variables and Standard Types

This chapter describes the global variables and the standard data types the C run-time library routines use.

Global Variables in Run-Time Routines

The C run-time library contains definitions for a number of variables and data types that library routines use. You can get access to these variables and data types by including the files that contain the following declarations or by declaring them in your programs.

_pgmptr

Format:

extern char far *_pgmptr;

The run-time variable *_pgmptr* points to the name of the executable file being run. Under DOS and in the DOS mode of OS/2, *_pgmptr* points to the program name which is stored in the environment segment. A typical example might be:

```
C:\TOOLS\TEST\PROG.EXE
```

In DOS and in the DOS mode of OS/2, the *_pgmptr* string is identical to the *argv*[0] argument passed to the main program.

In OS/2 mode, the string pointed to by *_pgmptr* will be the copy of the ProgPtr argument passed to the DOSEXECPGM function. If the program was called by CMD.EXE, it is a fully-qualified path string such as the example given above. If the program in question was called by another user program, *_pgmptr* points to the string that the other user program passed to DOSEXECPGM for the ProgPtr argument. Note that this is not necessarily the same as the *argv*[0] argument passed to the main program, although it may be depending on how the program was called. A program is always able to call a copy of itself using the *_pgmptr* string.

Format:

unsigned int _amblksiz;

You can use the *_amblksiz* variable to control the amount of storage space in the heap that C uses for dynamic storage. This variable is declared in the include file **malloc.h**.

The first time your program calls one of the dynamic storage allocation functions such as **calloc** or **malloc**, it asks the operating system for an initial amount of heap space that is typically much larger than the amount of storage that **calloc** or **malloc** request. This amount is shown by *_amblksiz*, whose default value is 8K bytes. The compiler reserves subsequent storage from this 8K bytes of storage, resulting in fewer calls to the operating system when the system is reserving many relatively small items. IBM C/2 calls the operating system again only if the amount of storage taken up by dynamic storage allocations exceeds the currently reserved space.

If the requested size in your C program is greater than *_amblksiz*, the system reserves multiple blocks, each of size *_amblksiz*, until it satisfies the request. Because the amount of heap space reserved is more than the amount requested, subsequent allocations can break up heap space. You can control this breaking up of heap space by using *_amblksiz* to change the default amount of storage to whatever value you would like, as in the following example:

```
_amblksiz = 2000;
```

Because the heap allocator always rounds the DOS request up to the nearest power of two greater than or equal to *_amblksiz*, the preceding statement causes the heap allocator to set aside storage in the heap in multiples of 2K bytes.

Adjusting *_amblksiz* affects only far heap allocation, that is, standard **calloc** and **malloc** calls in compact, large, and huge models, and **_fmalloc** calls in small and medium models. It has no effect on **halloc** or **_nmalloc** in any model.

daylight, timezone, tzname

Format:

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

Several time and date functions use the *daylight*, *timezone*, and *tzname* variables to make local time adjustments. The declarations for *daylight*, *timezone*, and *tzname* are in the **time.h** include file. The setting of an environment variable named TZ determines the values of these variables.

You can control local time adjustments by setting the TZ environment variable. The value of the environment variable TZ must be a 3-letter time zone, followed by a number, possibly signed, giving the difference in hours between Greenwich Mean Time and local time. A positive value for TZ denotes time zones west of the Greenwich meridian, and a negative number denotes time zones east of the Greenwich meridian. A 3-letter Daylight Saving Time zone can follow the number. For example, the command

```
SET TZ=EST5EDT
```

specifies that the local time zone is EST (Eastern Standard Time), that local time is 5 hours earlier than Greenwich Mean Time, and that Daylight Saving Time (EDT) is in effect. Omitting the Daylight Saving Time zone, as shown below, means that the program is to make no correction for Daylight Saving Time.

```
SET TZ=EST5
```

When you call the **ftime** or **localtime** function, the TZ setting determines the values of the three variables *daylight*, *timezone*, and *tzname*. The *daylight* variable receives a nonzero value if a Daylight Saving Time zone is present in the TZ setting; otherwise, *daylight* is zero. The *timezone* variable is assigned the difference in seconds (calculated by converting the hours given in the TZ setting) between Greenwich Mean Time and local time. The first element of the *tzname* variable is the string value of the 3-letter time zone from the

TZ setting; the second element is the string value of the Daylight Saving Time zone. If the Daylight Saving Time zone is omitted from the TZ setting, *tzname*[1] is an empty string.

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

```
EST5EDT
```

The **ftime** and **localtime** functions call another function, **tzset**, to assign values to the three global variables from the TZ setting. You can also call **tzset** directly if you like. See the **tzset** function in Chapter 5, "Library Routines," for details.

_doserrno, errno, sys_errlist, sys_nerr

Format:

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

The **perror** function uses the *errno*, *sys_errlist*, and *sys_nerr* variables to print error information. The include file **stdlib.h** contains the declarations for these variables. When an error occurs in a system-level call, the system sets the *errno* variable to an integer value to reflect the type of error. The **perror** function uses the *errno* value to look up (index) the corresponding error message in the *sys_errlist* table. The number of elements in the *sys_errlist* array defines the value of the *sys_nerr* variable.

The *errno* values for a DOS system are a subset of the values for *errno* for XENIX systems. Thus, the value assigned to *errno* in case of error does not necessarily correspond to the actual error code returned by a DOS system call. Instead, the system maps DOS error codes onto the **perror** values. If you want to access the DOS error code, you can use the *_doserrno* variable. When an error occurs in a system call, the *_doserrno* variable contains the error code returned by the corresponding DOS system call. (See the *IBM Disk Operating System Technical Reference* book for details about DOS error returns.)

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

Format:

int *_fmode*;

The *_fmode* variable controls the default file translation mode. The **stdlib.h** include file contains the declaration of *_fmode*. By default, the value of *_fmode* is zero, causing the system to translate files in text mode (unless specifically opened or set to binary mode). When you set *_fmode* to O_BINARY, the default mode is binary. Set *_fmode* to O_BINARY by linking with BINMODE.OBJ or by assigning it the value O_BINARY. See "Binary and Text Modes" in this book for a discussion of file translation modes and the use of the *_fmode* variable.

_osmajor, _osminor, _osmode

Format:

unsigned char *_osmajor*;
unsigned char *_osminor*;
unsigned char *_osmode*;

The *_osmajor* and *_osminor* variables provide information about the version number of DOS currently in use. The **stdlib.h** include file contains their declarations. The *_osmajor* variable holds the "major" version number. For example, under DOS Version 3.30, *_osmajor* is equal to 3.

The *_osminor* variable stores the "minor" version number. For example, under DOS Version 3.30, *_osminor* is 3.

These variables can be useful when you want to write code to run on different versions of DOS. For example, you can test the *_osmajor* variable before making a call to **sopen**. If the major version number of DOS is earlier than 3.00, use **open** instead of **sopen**.

The global variable *_osmode* is defined for DOS operation as the manifest constant DOS_MODE (defined in **stdlib.h**). Under OS/2, the *_osmode* variable stores the prevailing addressing mode. It assumes one of the values DOS_MODE or OS2_MODE. These constants, defined in **stdlib.h**, match the values returned from the OS/2 system call DOSGETMACHINEMODE.

Format:

char **environ[ ];*
unsigned int *_psp;*

The *environ* and *_psp* variables provide access to storage areas containing process-specific information. The **stdlib.h** include file contains declarations for both variables.

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

NAME = *string*

where NAME is an environment variable and *string* contains the value of that variable. The *string* can be empty. The system takes the initial environment settings from the DOS environment when the program runs.

The **getenv** and **putenv** routines use the *environ* variable to get access to and modify the environment table. You call **putenv** to add or delete environment settings. When you do this the environment table changes in size, and its location in storage can change, depending on the storage requirements of the program. The routine adjusts the *environ* variable in these cases so that it always points to the correct table location.

The *_psp* variable contains the segment value of the Program Segment Prefix (PSP) for the process. The Program Segment Prefix contains the performance information about the process, such as a copy of the command line that called the process and the return address for the command that ends or interrupts the process. See the *IBM Disk Operating System Technical Reference* book for details. You can use the *_psp* variable to form a **long** pointer to the Program Segment Prefix. The segment value is *_psp* and the offset value is 0.

The *_psp* variable is undefined under OS/2. Processes running under OS/2 have no PSP.

Standard Data Types in Run-Time Routines

A number of run-time library routines use data structures that are defined in include files. The following list describes each structure type and names the include file that defines it. For a listing of each structure definition, see the appropriate include file in Chapter 4, "Include Files."

Standard Type	Description
complex	The *complex* structure, defined in **math.h**, stores the real and imaginary parts of a complex number and is used by the **cabs** function.
DOSERROR	The DOSERROR structure, defined in **dos.h**, stores values returned by the DOS system call 59H (available under DOS Version 3.30 but not under OS/2).
exception	The *exception* structure, defined in **math.h**, stores error information for math routines and is used by the **matherr** routine.
FILE	The FILE structure, defined in **stdio.h**, is the structure used in all stream input and output operations. The fields of the FILE structure store information about the current state of the stream.
jmp_buf	The *jmp_buf* data type, declared in **setjmp.h**, is an array rather than a structure. It defines the buffer used by the **setjmp** and **longjmp** routines to save and restore the program environment.
REGS	The REGS data type, defined in **dos.h**, is a union rather than a structure. It stores byte-and word-register values to be passed to and returned from calls to the DOS interface functions. It is not applicable under OS/2.
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 (**int86x**, **intdosx**, and **segread**) that require segment register values. It is not applicable under OS/2.

stat	The *stat* structure, defined in **sys\stat.h**, contains file status information returned by the **stat** and **fstat** routines.
timeb	The *timeb* structure, defined in **sys\timeb.h**, is used by the **ftime** routine to store the current system time in four fields (*time*, *millitm*, *timezone*, and *dstflag*).
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 file-modification times used by the **utime** function to change file-modification dates.

Chapter 3. Run-Time Routines by Category

This chapter describes the major categories of routines in the IBM C/2 run-time library. The discussions of these categories give a brief overview of the capabilities of the run-time library. For a complete description of the syntax and use of each routine, see Chapter 5, "Library Routines."

Buffer Manipulation

The buffer manipulation routines are useful for working character-by-character with areas of storage. Buffers are arrays of characters (bytes). However, unlike strings, they do not usually end with a null character (\0). The buffer manipulation routines always take a length or count argument.

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

Routine	Use
memccpy	Copies characters from one buffer to another, until it copies a given character or a specified number of characters.
memchr	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer.
memcmp	Compares a specified number of characters from two buffers.
memcpy	Copies a specified number of characters from one buffer to another.
memset	Uses a given character to initialize a specified number of bytes in the buffer.
movedata	Copies a specified number of characters from one buffer to another, even when buffers are in different segments.

memicmp	Compares specified number of characters from two buffers without regard to case.

Character Classification and Conversion

The character classification and conversion routines let you test individual characters and convert characters between uppercase and lowercase. The classification routines identify a character by looking it up in a table of classification codes. Using these routines is generally faster than writing an equivalent test expression such as:

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

to classify a character.

Routine	Use
isalnum	Tests for an alphanumeric character.
isalpha	Tests for an alphabetic character.
isascii	Tests for an ASCII character.
iscntrl	Tests for a control character.
isdigit	Tests for a decimal digit.
isgraph	Tests for printable characters except for blanks (ASCII 32).
islower	Tests for a lowercase character.
isprint	Tests for a printable character.
ispunct	Tests for a punctuation character.
isspace	Tests for a white-space character.
isupper	Tests for an uppercase character.
isxdigit	Tests for a hexadecimal digit.
toascii	Converts a character to ASCII code.
tolower	Tests a character and converts to lowercase if uppercase.
toupper	Tests a character and converts to uppercase if lowercase.
_tolower	Converts a character to lowercase (unconditional).
_toupper	Converts a character to uppercase (unconditional).

You can use the **tolower** and **toupper** routines both as functions and as macros. The remainder of the routines in this category are only macros. The **ctype.h** include file contains definitions of all character classification and conversion macros. You must include this file or the macros in your program remain undefined.

The **toupper** and **tolower** macros evaluate their arguments twice. Arguments with side effects give incorrect results when you use these macros. Use the function versions of these routines instead.

The IBM C/2 compiler uses macro versions of **tolower** and **toupper** by default when you include **ctype.h**. To use the function versions instead, you must give **#undef** preprocessor directives for **tolower** and **toupper** *after* the **#include** directive for **ctype.h** but *before* you call the routines. This procedure removes the macro definitions and causes all occurrences of **tolower** and **toupper** to be treated as function calls to the **tolower** and **toupper** library functions.

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

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

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

An **#ifndef** block encloses the declarations of **tolower** and **toupper** in **stdlib.h**. The compiler processes these definitions only if the corresponding identifier (**toupper** or **tolower**) is not defined.

Data Conversion

The data conversion routines convert numbers to strings of ASCII characters and the reverse. These routines are functions, defined in the include file **stdlib.h**. The one exception is **atof**, defined in **math.h**. The **atof** function converts a string to a floating-point value.

Routine	Use
atof	Converts a string to a **float**.
atoi	Converts a string to an **int**.
atol	Converts a string to a **long**.
ecvt	Converts a **double** to a string.
fcvt	Converts a **double** to a string.
gcvt	Converts a **double** to a string and stores it in a buffer.
itoa	Converts an **int** to a string.
ltoa	Converts a **long** to a string.
strtod	Converts a string to a **double**.
strtol	Converts a string to a **long** decimal integer that is equal to a number with the specified radix.
ultoa	Converts an **unsigned long** to a string.

Directory Control

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

The declarations of the directory routines, which are functions, are in the include file **direct.h**.

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

File Handling

The file handling routines work on a file designated by a pathname or file handle. They change or give information about the designated file. All of these routines except **fstat** and **stat** are declared in the include file IO.H. The declarations of the **fstat** and **stat** functions are in **sys\stat.h**. The declarations of the **remove** and **rename** functions are in **stdio.h**.

Routine	Use
access	Checks a file permission setting.
chmod	Changes a file permission setting.
chsize	Changes a file size.
filelength	Checks a file length.
fstat	Gets a file status information on file handle.
isatty	Checks for a character device.
locking	Locks areas of a file.
mktemp	Creates a unique filename.
remove	Deletes a file.
rename	Renames a file.
setmode	Sets a file translation mode.
stat	Gets file-status information on a named file.
umask	Sets the default permission mask.
unlink	Deletes a file.

The **access**, **chmod**, **rename**, **remove**, **stat**, and **unlink** routines operate on files specified by a pathname or filename.

The **chsize**, **filelength**, **fstat**, **isatty**, **locking**, and **setmode** routines work with files designated by a file handle. The **locking** routine locks a region of a file against access by other users.

The **mktemp** and **umask** routines have slightly different functions than the above routines. The **mktemp** routine creates a unique filename. Programs can use **mktemp** to create unique filenames 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 cancel the permission setting given in the **open** or **creat** call for the new file.

Input and Output

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

- Stream input/output
- Low-level input/output
- Console and port input/output.

The maximum number of file handles on DOS is 20, but on OS/2 you set the maximum using the OS/2 function DOSSETMAXFH. The C libraries impose a maximum of 40 file handles.

Stream Routines

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

When you open a file for input or output using the stream functions, the system associates the opened file with a structure of type FILE, defined in **stdio.h**, containing basic information about the file. The system returns a pointer to the FILE structure when you open the stream. This pointer, called the *stream pointer* or just *stream*, refers to the file in subsequent operations.

The stream functions can provide for optionally buffered and formatted input and output. When you direct a stream to a buffer, the system collects data read from or written to the stream in an intermediate storage location called a *buffer*. During a write operation, the system writes the contents of the output buffer to the appropriate final location only after the buffer is full, when the stream is closed, or when the program ends normally. To carry out this operation is to *flush* the buffer. During a read operation, the system places a block of data in the input buffer and reads the data from the buffer. The system transfers the next block of data into the buffer only after the buffer is empty.

Buffering lets input or output proceed efficiently because the system can transfer a large block of data in a single operation rather than performing an input or output operation each time it reads a data item from or writes one to a stream. However, if a program ends abnormally, the system might not flush the output buffers and could lose data.

Routine	Use
clearerr	Clears the error indicator for a *stream*.
fclose	Closes a *stream*.
fcloseall	Closes all open *streams*.
fdopen	Opens a *stream* using a *handle*.
feof	Tests for an end-of-file on a *stream*.
ferror	Tests for an 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).
fgets	Reads a string from *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 items from a *stream*.
freopen	Reassigns a FILE pointer.
fscanf	Reads formatted data from a *stream*.
fseek	Repositions the file pointer to a given location.
ftell	Gets the current file pointer position.
fwrite	Writes fixed-length data items to **stdout**.
getc	Reads a character from a *stream* (macro version).
getchar	Reads a character from **stdin** (macro version).
gets	Reads a line from **stdin**.
getw	Reads a binary **int** from a *stream*.
printf	Writes formatted data to **stdout**.
putc	Writes a character to a *stream* (macro version).
putchar	Writes a character to **stdout** (macro version).
puts	Writes a line to a *stream*.
putw	Writes a binary **int** to a *stream*.
rewind	Repositions file pointer to beginning of a *stream*.
rmtmp	Removes temporary files created by a **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	Produces a temporary file name in a given directory.
tmpfile	Creates a temporary file.
tmpnam	Produces a temporary file name.
ungetc	Places a character in the buffer.
vfprintf	Writes formatted data to a *stream*.
vprintf	Writes formatted data to **stdout**.
vsprintf	Writes formatted data to a string.

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

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

Opening a Stream: You must open a stream with the **fdopen**, **fopen**, or **freopen** function before you can perform input or output on that stream. You can open a stream for reading, writing, or both. You can open a stream in text mode or binary mode.

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

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

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

Predefined Stream Pointers: stdin, stdout, stderr, stdaux, stdprn: When a program begins to run, the system automatically opens five streams. These streams are the standard input, standard output, standard error, standard auxiliary, and standard print streams. By default, the standard input, standard output, and standard error streams refer to the keyboard and screen. Whenever a program expects input from the standard input stream, it receives that input from the keyboard. Similarly, a program that writes to the standard output stream displays data on the screen. The system sends error messages produced by the library routines to the standard error stream. The error messages interrupt the standard output stream and appear on the screen.

The assignment of the standard auxiliary stream and the standard print stream depends on the machine setup. These streams usually refer to an auxiliary port and a printer, respectively, but they might not have a device attached on a particular system. Be sure to check your machine setup before using these streams.

When you use the stream functions, you can refer to the standard input, standard output, standard error, standard auxiliary, and standard print streams by using the following predefined **FILE** pointers.

Stream	**Device**
stdin	Standard input
stdout	Standard output
stderr	Standard error
stdaux	Standard auxiliary
stdprn	Standard print.

You can use these pointers in any function that requires a stream pointer as an argument. Some functions, such as **getchar** and **putchar**, 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. Pointers **stdaux** and **stdprn** are not predefined under OS/2.

You can use the DOS redirection symbols (<,>, or >>) or the pipe symbol (|) to redefine the standard input and standard output for a particular program. See the *IBM Disk Operating System Technical Reference* book for a complete discussion of *redirection* and *pipes*.

For example, if you run a program and redirect its output to a file named RESULTS, the program writes to the RESULTS file each time the standard output is specified in a write operation. You do not change the program when you redirect the output. You change only the file associated with **stdout** for a single run of the program.

You can redefine **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** to refer to a disk file or to a device. The **freopen** routine reassigns the stream. See the **freopen** function in Chapter 5, "Library Routines," for a description of this option.

Note: You cannot redirect **stderr** (the standard error stream) at the DOS command level, though you can from OS/2. For example, use the command line

```
cl main.c 2>errmsg
```

in OS/2 to redirect the error stream from the compiler to a file called ERRMSG.

Controlling Stream Buffering

By default, the system buffers any files that you open by using the stream functions, except for the preopened streams **stderr**, **stdaux**, **stdin**, **stdout**, and **stdprn**. The **stderr** and **stdaux** streams are unbuffered, unless they are being used by **printf** or **scanf**, which assign a temporary buffer. These two streams can also be buffered using **setbuf** or **setvbuf**. Streams **stdin**, **stdout**, and **stdprn** are *buffered*. The compiler flushes the buffer whenever it is full, or whenever the function causing I/O ends.

By using the **setbuf** or **setvbuf** functions, you can cause streams to be unbuffered or you can associate a buffer with an unbuffered stream. Buffers reserved by the system are not accessible, but you can name buffers reserved with **setbuf** or **setvbuf**. You can manipulate them as if they were variables. Buffers can have any size. If you use **setbuf**, it sets the manifest size for the constant BUFSIZ in **stdio.h**. If you use **setvbuf**, you can set the size of the buffer yourself. For more information about these two functions, see **setbuf** and **setvbuf** in Chapter 5, "Library Routines."

The system automatically flushes buffers when they reach the size of BUFSIZ, when the system closes the associated file, or when a program ends normally. You can flush buffers at other times with the

fflush and **flushall** routines. The **fflush** routine flushes a single specified stream, while **flushall** flushes all open, buffered streams.

Closing Streams

The **fclose** and **fcloseall** functions close a stream or streams. The **fclose** routine closes a single specified stream. The **fcloseall** routine closes all open streams except **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. If your program does not explicitly close a stream, the system automatically closes the stream when the program ends. It is good practice to close a stream when you finish with it.

Reading and Writing Data

The stream functions let you transfer data in a variety of ways. You can read and write binary data or specify the reading or writing of characters, lines, or more complicated formats. A summary of the stream functions for reading and writing data is at the beginning of this section. For a full description of each function, see Chapter 5, "Library Routines."

Reading and writing operations on streams begin at the current position in the stream, known as the *file pointer* for the stream. After a reading or writing operation, the functions change the file pointer to reflect the new position of the file pointer. For example, if you read a single character from a stream, the function increases the file pointer by 1 byte. The next operation now begins at the first unread character. If you open a stream to add something, the system automatically positions the file pointer at the end of the file before each writing operation. A new-line character is not required at the end of a text stream. A text line containing only a single space character plus a terminating new-line character is not converted on input to a line consisting only of the terminating new-line character; the space is left in the string. NUL characters are never appended to data written to binary streams.

The **feof** macro detects an end-of-file in a stream. After you set the end-of-file indicator, it remains set until you close the file, return to the beginning of the file, or call the **clearerr** or **rewind** functions.

You can position the file pointer anywhere in a file with the **fseek** function. The next operation takes place at the position you specified.

The **rewind** function positions the file pointer at the beginning of the file. Use the **ftell** function to tell the current position of the file pointer.

Streams associated with a device, such as a screen, do not have file pointers. You cannot get random access to data going to a screen. Routines that use file pointers have undefined results if you use them on a stream associated with a device.

Detecting Errors

When an error occurs in a stream operation, the system sets an error indicator for the stream. You can use the **ferror** macro to test the error indicator and tell whether an error has occurred. After an error occurs, the error indicator for the stream remains set until you close the stream, return to the beginning of the stream, or explicitly clear the error indicator by calling the **clearerr** or **rewind** function.

Low-Level Routines

The *low-level input* and *output* routines do not buffer or format data. They directly call input and output capabilities of the operating system. These routines let you get access to files and peripheral devices at a more basic level than the stream functions.

When you open a file with a low-level routine, the compiler associates a file handle with the opened file. This handle is an integer value used to refer to the file in subsequent operations.

CAUTION:
Stream routines and low-level routines are generally incompatible. Use either stream or low-level functions consistently on a given file. Because stream functions are buffered and low-level functions are not, attempting to get access to the same file or device by two different methods causes confusion and can result in the loss of data in buffers.

Routine	Use
close	Closes a file.
creat	Creates a file.

dup	Creates a second *handle* for a file.
dup2	Reassigns a file *handle*.
eof	Tests for an end-of-file.
lseek	Repositions the 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 the current file pointer position.
write	Writes data to a file.

Low-level input and output calls do not buffer or format data. Use a file handle to refer to files opened by low-level calls. This is an integer value that the operating system uses to refer to the file. Use the **open** function to open files. On DOS Version 3.30, you can use **sopen** to open a file with file-sharing attributes.

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

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

Opening a File

You must open a file with the **open**, **sopen**, or **creat** function before you can perform input and output with the low-level functions on that file. You can open the file for reading, writing, or both. You can open the file in either text mode or binary mode. You must include the file **fcntl.h** when opening a file. The **fcntl.h** file contains definitions for flags that the **open** function uses. In some cases, you must also include the files **sys\types.h** and **sys\stat.h**. For details see the **open** routine in Chapter 5, "Library Routines."

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

Predefined Handles

When a program begins to run, the system assigns five file handles, corresponding to the standard input, standard output, standard error, standard auxiliary, and standard print streams. By using the following predefined handles, a program can call low-level functions to get access to the standard input, standard output, standard error, standard auxiliary, and standard print streams described with the stream functions in this chapter.

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

You can use these file handles in your program without opening the associated files. The compiler opens the files when the program begins, as the following short program shows. This example uses the **fileno** function to print the file handle values assigned to the standard input, standard output, standard error, standard auxiliary, and standard print streams:

```
#include <stdio.h>

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

Output:

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

As with the stream functions, you can use redirection and pipe symbols when you run your program to redirect the standard input and standard output. The **dup** and **dup2** functions let you assign mul-

tiple handles for the same file. You can use these functions to associate the predefined file handles with different files.

Note: You cannot redirect **stderr** (the standard error stream) at the DOS command level, though you can under OS/2. For example, use the command line

```
cl main.c 2>errmsg
```

in OS/2 to redirect the error stream from the compiler to a file called ERRMSG.

Reading and Writing Data

Two basic functions, **read** and **write**, perform input and output. As with the stream functions, reading and writing operations always begin at the current position in the file. You must update the current position each time a read or write operation takes place.

You can use the **eof** routine to test for an end-of-file condition. Low-level input/output routines set the **errno** variable when an error occurs. You can use the **perror** function to print information about input/output errors.

You can position the file pointer anywhere in a file with the **lseek** function. The next operation takes place at the position you specified. Use the **tell** function to tell the current position of the file pointer.

Devices, such as the screen or a printer, do not have file pointers. The **lseek** and **tell** routines have undefined results if you use them on a handle associated with a device.

Closing Files

The **close** function closes an open file. The system automatically closes all open files when a program ends. However, it is good practice to close a file when you finish with it.

Keyboard and Port I/O Routines

The *keyboard and port input/output* routines are an extension of the stream routines. They let you read from a keyboard or read from or write to an input/output port, such as a printer port. The port input/output routines read and write data in bytes. Some additional options are available with keyboard input/output routines. For example, your program can detect whether a user has typed a character on the keyboard. You can also choose between echoing characters to the screen as the system reads them or reading characters without echoing. The routines are:

Routine	Use
cgets	Reads a string from the keyboard.
cprintf	Writes formatted data to the screen.
cputs	Writes a string to the screen.
cscanf	Reads formatted data from the keyboard.
getch	Reads a character from the keyboard.
getche	Reads a character from the keyboard and echoes it.
inp	Reads from a specified input port.
kbhit	Checks for a keystroke at the keyboard.
outp	Writes to a specified output port.
putch	Writes a character to the screen.
ungetch	Pushes the last character back to the keyboard again so that it becomes the next character read.

The keyboard and port input/output routines are functions in the include file **conio.h**. These functions perform reading operations from your keyboard, writing operations on your screen, or reading or writing operations on a specified port. The **cgets**, **cscanf**, **getch**, **getche**, and **kbhit** functions take input from the keyboard, while **cprintf**, **cputs**, **putch**, and **ungetch** write to the screen. Redirecting the standard input or standard output streams from the command line redirects the input or output of these functions.

You do not have to open or close the port for the keyboard or screen before you perform an input or output operation. Consequently, there are no open or close routines in this category. The port input/output routines **inp** and **outp** read 1 byte at a time from or write 1 byte at a time to the specified port. The keyboard and screen input/output routines allow the reading and writing of strings (**cgets** and **cputs**), for-

matted data (**cscanf** and **cprintf**), and characters. Several options are available for reading and writing characters.

The **putch** routine writes a character to the screen. The **getch** and **getche** routines read a character from the keyboard. **Getche** echoes the character back to the screen, and **getch** does not. The **ungetch** routine pushes the last character read back to the keyboard. The next read operation on the keyboard begins with that character, the last character typed.

The **kbhit** routine tells when a key has been struck at the keyboard. This routine lets you test for keyboard input before you try to read.

Note: The keyboard and screen input/output routines use the corresponding DOS system calls to read and write characters. See the *IBM Disk Operating System Technical Reference* book for the details of the specific system calls. Under OS/2, you may not redirect data to or from other files using these routines. The keyboard and screen input/output routines always use the console.

Math

The math routines let you perform common mathematical calculations. All math routines (except **matherr**, the error-handling function) work with floating-point values and thus require floating-point support. See the section on "Floating-Point Support" in Chapter 1 of this book for additional information. The include file **math.h** gives function declarations for the math routines, except for **_clear87**, **_control87**, **_fpreset**, and **_status87**, whose definitions are in the **float.h** include file.

Routine	Use
acos	Calculates an arc cosine.
asin	Calculates an arc sine.
atan	Calculates an arc tangent of one argument.
atan2	Calculates an arc tangent of two arguments.
bessel	Calculates bessel functions.
cabs	Finds the absolute value of a complex number.
ceil	Finds the integer ceiling of a **double**.

_clear87	Gets and clears a floating-point status word.
_control87	Gets an old floating-point control word and sets a new control-word value.
cos	Calculates a cosine.
cosh	Calculates a hyperbolic cosine.
dieeetomsbin	Converts an IEEE double-precision format to a binary double format.
dmsbintoieee	Converts a binary double format to an IEEE double-precision format.
exp	Calculates an exponential function.
fabs	Finds an absolute value of a **double**.
fieeetomsbin	Converts an IEEE single-precision format to binary format (**float**).
floor	Finds an integer floor of a **double**.
fmod	Finds a remainder.
fmsbintoieee	Converts a binary format (**float**) to IEEE single-precision format.
_fpreset	Reinitializes the floating-point math package.
frexp	Breaks down a **double** into a mantissa and exponent.
hypot	Calculates an hypotenuse.
ldexp	Calculates *x* times a power of 2.
log	Calculates a natural logarithm.
log10	Calculates a base 10 logarithm.
matherr	Handles math errors.
modf	Breaks down a **double** into an integer and fractional parts.
pow	Calculates a power.
sin	Calculates a sine.
sinh	Calculates a hyperbolic sine.
sqrt	Finds a square root.
_status87	Gets the floating-point status word.
tan	Calculates a tangent.
tanh	Calculates a hyperbolic tangent.

The math functions call the **matherr** routine when errors occur. This routine is defined in the library, but you can redefine it if you want different error-handling procedures. When you define the **matherr** function, it must conform to the specifications for the **matherr** function in Chapter 5, "Library Routines."

You need not supply a definition for **matherr**. If no definition is present, the system returns the default error for each routine. See the reference page for each routine in Chapter 5, "Library Routines," for a description of its error return values.

Reserving Storage

The storage allocation routines let you reserve, free, and reallocate blocks of storage. The include file **malloc.h** contains the declarations of these functions.

Routine	Use
alloca	Reserves storage from the stack.
calloc	Reserves storage for an array.
_expand	Expands or contracts a block of storage without moving its location.
_ffree	Frees a block reserved by **_fmalloc**.
_fmalloc	Reserves a block of storage outside the default data segment, returns a far pointer.
free	Frees a reserved block.
_freect	Returns the approximate number of items of a given size that the compiler can reserve.
_fmsize	Returns the size of a storage block pointed to by a **far** pointer.
halloc	Reserves a **huge** array.
hfree	Frees a block reserved by **halloc**.
malloc	Reserves a block.
_memavl	Reports the approximate number of bytes available for reserving in the heap in memory.
_msize	Returns the size of the block reserved by **calloc**, **malloc**, or **realloc**.
_nfree	Frees a block reserved by **_nmalloc**.
_nmalloc	Reserves a block of storage in the default data segment, returns a near pointer.
_nmsize	Returns the size of a storage block pointed to by a near pointer.
realloc	Changes the size of a previously reserved block of storage.
sbrk	Resets the break value.
stackavail	Returns the size of the stack space available for reserve with **alloca**.

The **calloc** and **malloc** routines reserve storage blocks. The **malloc** routine reserves a given number of bytes; the **calloc** routine reserves and initializes to 0 an array with elements of a given size. The rou-

tines **_fmalloc** and **_nmalloc** are similar to **malloc**, except that **_fmalloc** and **_nmalloc** let you reserve blocks of bytes while overcoming the addressing limitations of the current storage model. The **halloc** routine performs essentially the same function as **calloc**, with the difference that **halloc** reserves space for huge arrays (those exceeding 64K bytes in size). Arrays allocated with **halloc** must satisfy the requirements for huge arrays, as outlined in "Creating Huge Model Programs" in the *IBM C/2 Compile, Link, and Run* book. The **realloc** and **_expand** routines change the size of an allocated block. The **_expand** function always attempts to change the size of an allocated block without moving its heap location. It expands the size of the block up to the size requested or as much as the current location allows, whichever is smaller. In contrast, **realloc** changes the location in the heap if there is not enough room. The **halloc** routine returns a huge pointer, **_fmalloc** returns a far pointer, and **_nmalloc** returns a near pointer. These routines all return a pointer to void; the space to which they point satisfies the alignment requirements for any type of object. Use a type cast on the return value to obtain the type of pointer you need.

When **_fmalloc** is called it allocates a segment from DOS, returns the requested amount of memory, and does heap management on the rest for subsequent calls to **_fmalloc**. When it runs out of memory on its current segment, it goes to DOS for another. While **_fmalloc** allocates memory from DOS, **_ffree** returns it to **_fmalloc**'s heap (the far heap). The **_fmalloc** routine attempts to allocate memory from the near heap in the default data segment as a last resort.

When an IBM C/2 program is loaded, it reduces its DOS allocated memory to:

> program size + global and static variables + stack + heap area for **_nmalloc**,

where variables + stack + heap area <=64K.

This overrides the specification in the program header which tells the loader to use all the memory. The extent to which the original allocation gets cut back is controlled by using the /CPARMAXALLOC link option described in *Compile, Link, and Run.*

The **_nmalloc** and **_fmalloc** routines exhibit performance differences. **_nmalloc** is the best for small memory allocation where total memory

allocation requirements are less than 64K. This amount is smaller depending on the number of external variables and the amount of runtime I/O. **_fmalloc** is best when total memory allocation requirements are greater than 64K but no single data object is greater than 64K. The **halloc** function is the slowest of all because it petitions DOS for every memory request. Select **halloc** as the function of choice when either you want data objects larger than 64K or you want to be certain you can free allocated memory back to DOS for subsequent program spawns.

The **free** routine (for **calloc**, **malloc**, and **realloc**), the **_ffree** routine (for **_fmalloc**), the **_nfree** routine (for **_nmalloc**), and the **hfree** routine (for **halloc**) all free storage that was previously reserved, making it available for later requests.

The **_freect** and **_memavl** routines tell you how much storage is available for dynamic storage allocation in the default data segment. The **_freect** function returns the approximate number of items of a given size for which the compiler can reserve storage. The **_memavl** function returns the total number of bytes available for allocation requests.

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

The **sbrk** routine is a low-level routine that reserves storage. It increases the break value of the program, letting the program take advantage of available unreserved storage.

CAUTION:
A program that uses the sbrk routine should not use the other routines that reserve storage although the compiler does not prohibit their use. Using sbrk to decrease the break value can cause later calls to other storage allocation routines to give unpredictable results.

The preceding routines all reserve storage dynamically from the heap. IBM C/2 also provides two storage functions, **alloca** and **stackavail**, for reserving space from the stack and determining the amount of available stack space. The **alloca** function reserves, from

the stack, the requested number of bytes that the compiler frees when control returns from the function calling **alloca**. The **stackavail** function lets your program know how much storage (in bytes) is available on the stack.

DOS Interface

These routines provide access to DOS system calls and interrupts. See the *IBM Disk Operating System Technical Reference* book for information on system calls and interrupts.

Routine	Use
bdos	Makes a DOS system call; uses only the DX and the AL registers.
dosexterr	Obtains register values from DOS system call function 59H.
FP_OFF	Returns offset portion of a **far** pointer.
FP_SEG	Returns segment portion of a **far** pointer.
int86	Calls 8086 software interrupt.
int86x	Calls 8086 software interrupt.
intdos	Makes a DOS system call; uses registers other than DX and AL.
intdosx	Makes a DOS system call; uses segment registers.
segread	Returns current values of segment registers.

The FP_OFF and FP_SEG routines let you easily get access to the segment and offset portions of a **far** pointer value. The FP_OFF and FP_SEG routines are macros defined in **dos.h**. The remaining routines are functions declared in **dos.h**.

The **dosexterr** function obtains and stores the register values returned by DOS system call 59H (extended error handling). This function is for DOS Version 3.30.

The **bdos** routine makes DOS calls that use either or both of the DX (DH/DL) or AL registers for arguments. However, do not use **bdos** to make system calls that return an error code in AX if the carry flag is set. The program cannot detect whether the carry flag is set, making it impossible to determine whether the value in AX is a legitimate value or an error value. In this case use the **intdos** routine instead, because it lets the program detect whether the carry flag is set. You can also use the **intdos** routine to call DOS calls that use registers other than DX and AL.

The **intdosx** routine is similar to the **intdos** routine, but you use it 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 a system call in a large model program. When calling **intdosx**, give an argument that specifies the segment values used in the call.

Use the **int86** routine to call DOS interrupts. The **int86x** routine is similar, but, like the **intdosx** routine, works with large model programs and **far** items as described in the preceding paragraph for **intdosx**.

The **segread** routine returns the current values of the segment registers. Use this routine with the **intdosx** and **int86x** routines to obtain the correct segment values.

Process Control

The term *process* refers to a program running under the operating system. A process consists of the code and data for the program and information about the status of the running program, such as the number of open files. Whenever you run a program at the DOS level, you start a process. In addition, you can start, stop, and manage processes from within a program by using the process control routines.

The process control routines let you:

- Identify a process by a unique number (**getpid**)
- Stop a process (**abort**, **exit**, and **_exit**)
- Handle an interrupt signal (**raise** and **signal**)
- Start a new process (the **exec** and **spawn** families of routines, plus the **system** routine).

The declarations for all process control functions except **raise** and **signal** are in the include file **process.h**. The declaration for the **raise** and **signal** function are in the **signal.h** include file.

Routine	Use
abort	Stops a process.
execl	Runs a child process, using an argument list.

execle	Runs a child process, using an argument list and a given environment.
execlp	Runs a child process, using the PATH variable and an argument list.
execlpe	Runs a child process, using the PATH variable and an argument list.
execv	Runs a child process, using an argument array.
execve	Runs a child process, using an argument array and a given environment.
execvp	Runs a child process, using the PATH variable and an argument array.
execvpe	Runs a child process, using the PATH variable, an argument array, and a given environment.
exit	Ends a process, flushing buffers and closing files.
_exit	Ends a process without flushing the buffers.
getpid	Gets a process identification number.
onexit	Runs functions at the normal or abnormal ending of a program.
raise	Sends a signal to a program.
signal	Handles an interrupt signal.
spawnl	Starts a child process, using an argument list.
spawnle	Starts a child process, using an argument list and a given environment.
spawnlp	Starts a child process, using the PATH variable and an argument list.
spawnlpe	Starts a child process, using the PATH variable, an argument list, and a given environment.
spawnv	Starts a child process, using an argument array.
spawnve	Starts a child process, using an argument array and a given environment.
spawnvp	Starts a child process, using the PATH variable and an argument array.
spawnvpe	Starts a child process, using the PATH variable, an argument array, and a given environment.
system	Runs a DOS command.

The **abort** and **_exit** functions immediately end a process without flushing the stream buffers. The **exit** function ends a process after flushing the stream buffers.

The **system** function runs a given DOS command. The **exec** and **spawn** functions start up a new process, called the *child process*. The difference between the **exec** and **spawn** routines is that the **spawn** routines

can return control from the child process to its caller (the *parent* process). Under DOS, both the parent process and the child process are present in memory unless you specify P_OVERLAY. Under OS/2, the parent and child may or may not both be in memory, if swapping is enabled.

Under DOS in the **exec** routines, the child process *overlays* the parent process. Returning control to the parent process is impossible unless an error occurs when you attempt to start running the child process. Under OS/2, **exec** is simulated. The child process does not actually overlay the parent in storage, but it is not possible to return control to the parent process, which ends as soon as the child starts.

There are eight forms each of the **spawn** and **exec** routines. The following table summarizes the differences between the forms. The function names are in the first column. The second column specifies whether the current PATH setting locates the file to be run as the child process.

The third column describes the method for passing arguments to the child process. Passing an argument list means that you list the arguments to the child process as separate arguments in the **exec** or **spawn** call. Passing an argument array means that the arguments are in an array and a pointer to the array is passed to the child process. Use the argument-list method when the number of arguments is constant or is known when you compile. Use the argument-array method when you cannot tell the number of arguments until you run the program.

The last column specifies if the child process inherits the environment settings of its parent or you must pass a pointer to a table of environment settings to set up a different environment for the child process.

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

Searching and Sorting

The run-time library has three search routines and one sort routine.

Routine	Use
bsearch	Performs a binary search.
lfind	Performs a linear search for a given value.
lsearch	Performs a linear search of an array for a given value. If **lsearch** does not find the value in the array, it adds it to the array.
qsort	Performs a quick-sort.

The **bsearch**, **lfind**, **lsearch** and **qsort** functions provide helpful binary search, linear search, and quick-sort utilities. The include file **search.h** contains the declarations for these functions.

Manipulating Strings

The declarations of the string functions are in the include file **string.h**. A wide variety of string functions is available in the run-time library. You can:

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

The following string functions are in **string.h**:

Routine	Use
strcat	Adds a string to a string.
strchr	Finds the first occurrence of a given character in a string.
strcmp	Compares two strings.
strcmpi	Compares two strings without regard to case.
strcpy	Copies one string to another.
strcspn	Finds the first occurrence of a character from a given character set in a string.
strdup	Reserves a space and copies a string.
strerror	Saves the system error message and optional user error messages in a string.
stricmp	Compares two strings without regard to case (identical to **strcmpi**).
strlen	Finds the length of a string.
strlwr	Converts a string to lowercase.
strncat	Adds *n* characters to the end of a string.
strncmp	Compares *n* characters of two strings.
strncpy	Copies *n* characters of one string to another.

strnicmp	Compares *n* characters of two strings without regard to case. (The "i" indicates this function is "case insensitive".)
strnset	Sets *n* characters of a string to a given character.
strpbrk	Finds the first occurrence of a character from one string in another.
strrchr	Finds the last occurrence of a given character in a string.
strrev	Reverses a string.
strset	Sets all characters of a string to a given character.
strspn	Finds the first occurrence of a character in a string that is not in a given string.
strstr	Finds the first occurrence of a given string in another string.
strtok	Finds the next token in a string.
strupr	Converts a string to uppercase.

All string functions work on character strings that end with a null escape sequence **\0**. When working with character arrays that do not end with a null character, you can use the buffer manipulation routines, described earlier in this chapter.

Time

The time functions let you obtain the current time, then convert and store it according to your particular needs. The current time is always the system time. The **time** and **ftime** functions return the current time as the number of seconds elapsed since Greenwich Mean Time, January 1, 1970. You can convert, adjust, and store this value in a variety of ways, using the **asctime**, **ctime**, **gmtime**, and **localtime** 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.

Routine	Use
asctime	Converts the time from a structure to a character string.
ctime	Converts the time from a long integer to a character string.
difftime	Computes the difference between two times.
ftime	Gets the current system time as a structure.
gmtime	Converts the time from an integer to a structure.

localtime	Converts the time from an integer to a structure with local correction.
time	Gets the current system time as a long integer.
tzset	Sets external time variables from the environment time variable.
utime	Sets the file modification time.

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

When you want to use **ftime** or **localtime** to make adjustments for local time, you must define an environment variable named TZ. See the discussion of *daylight*, *timezone*, and *tzname* in Chapter 2, "Global Variables and Standard Types." TZ is also described on the **tzset** reference page in Chapter 5, "Library Routines."

Variable-Length Argument Lists

The **va_arg**, **va_end**, and **va_start** routines are macros that provide a portable way to get access to the arguments of a function when the function takes a variable number of arguments. The macro defined in **stdarg.h** conforms to the proposed ANSI C standard. For more information, see the discussion of the **va_arg-va_start** in Chapter 5, "Library Routines."

Routine	Use
va_arg	Retrieves an argument from an argument list.
va_end	Sets the argument pointer to the end of an argument list.
va_start	Sets the pointer to the beginning of an argument list.

Miscellaneous Routines

The miscellaneous category covers a number of commonly-used routines that do not fit easily into any of the other categories. The declarations for all routines except **assert**, **longjmp**, and **setjmp** are in **stdlib.h**.

Routine	Use
abs	Finds the absolute value of an **int**.
assert	Tests for a logic error.
getenv	Gets the value of an environment variable.
labs	Finds the absolute value of a **long**.
longjmp	Restores a saved stack environment.
perror	Prints an error message.
putenv	Adds or changes the value of an environment variable.
rand	Gets a pseudo-random number.
setjmp	Saves a stack environment.
srand	Initializes a pseudo-random number sequence.
swab	Swaps bytes of data.

The **assert** routine is a macro defined in **assert.h**. The declarations for the **setjmp** and **longjmp** functions are in **setjmp.h**.

The **abs** and **labs** functions return the absolute value of an **int** and a **long** value, respectively.

Use the **assert** macro to test for program logic errors; it prints a message when a given assertion fails to hold true. Defining the identifier NDEBUG removes from the source file any occurrences of **assert**. This lets you turn off assertion-checking without changing the source file.

The **getenv** and **putenv** routines provide access to the environment table. The global variable *environ* also points to the environment table. It is recommended that you use the **getenv** and **putenv** routines to get access to and change the environment settings instead of getting access to the environment table directly.

The **perror** routine prints the system error message or messages that you supply for the last system-level call that produces an error. The declaration of the **perror** function is in **stdlib.h**. The routine obtains

the error number from the *errno* variable and the system message from the **sys_errlist** array. The *errno* variable is guaranteed to have been set only by those routines that explicitly mention the *errno* variable in the "Remarks" sections of Chapter 5, "Library Routines."

The **rand** and **srand** functions initialize and generate a pseudo-random number sequence of integers.

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

The **swab** routine (also declared in **stdlib.h**) swaps bytes of binary data. Use it to prepare data for transfer to a system that uses a different byte order.

Chapter 4. Include Files

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

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

Two sets of routine declarations are provided in each include file. The first set declares both the routine return type and the argument type list for the routine. This set is included only when you enable argument type-checking by defining LINT_ARGS, as described in "Argument Type-Checking" in Chapter 1, "About the IBM C/2 Library." The second set of declarations declares only the routine return type. This set is included when argument type-checking is not enabled.

The include files were named and organized to:

- Maintain compatibility of include file names with the developing ANSI standard
- Reflect the logical categories of run-time routines (for example, placing declarations for all storage allocation routines in one file, **malloc.h**)
- Require the inclusion of the minimum number of files to use a given routine.

Occasionally, these goals conflict. For example, the **ftime** routine uses the structure type **timeb**. The **timeb** structure type is defined in the include file **sys\timeb.h** on some systems. To maintain compatibility, the same include file is used on DOS. To reduce the number of

required include files when using **ftime**, the **ftime** routine is declared in **sys\timeb.h**, even though the declarations for most of the other time routines are in **time.h**.

assert.h

The **assert.h** include file defines the **assert** macro. You must include the **assert.h** file when you use **assert**.

The definition of **assert** is in an **#ifndef** preprocessor block. If you have not defined the identifier NDEBUG through a **#define** directive or on the compiler command line, the **assert** macro tests a given expression (the "assertion"). If the assertion is false, the system prints a message and the program ends.

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

conio.h

The **conio.h** include file contains routine declarations for all of the screen and port input/output routines listed below:

cgets	**cscanf**	**inp**	**putch**
cprintf	**getch**	**kbhit**	**ungetch**
cputs	**getche**	**outp**	

ctype.h

The **ctype.h** include file defines macros and constants and declares a global variable used in character classification. The macros defined in **ctype.h** are listed below. You must include **ctype.h** when using these macros or the macros are undefined.

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

The **toupper** and **tolower** macros are defined as conditional operations. These macros evaluate their argument twice and produce unexpected results for arguments with side effects. To overcome this problem, you can remove the macro definitions of **toupper** and **tolower** and use the routines by the same names. For more information about conditional operations, see "Character Classification and Conversion" in Chapter 3, "Run-Time Routines by Category." Declarations for the routine versions of **tolower** and **toupper** are given in **stdlib.h**.

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

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

direct.h

The **direct.h** include file contains routine declarations for the four directory control routines (**chdir**, **getcwd**, **mkdir**, and **rmdir**).

dos.h

The **dos.h** include file contains macro definitions, routine declarations, and type definitions for the DOS interface routines.

The FP_SEG and FP_OFF macros are defined to obtain the segment and offset portions from a **far** pointer value. You must include **dos.h** when using these macros or they remain undefined. The following routines are declared in **dos.h**:

bdos **int86** **intdos** **segread**
dosexterr **int86x** **intdosx**

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

The DOSERROR structure is defined to hold error values returned by DOS system-call 59H (available under DOS 3.30 , but not under OS/2).

WORDREGS, BYTEREGS, REGS, SREGS, and DOSERROR are tags, not **typedef** names. (For more information about type definitions, tags, and **typedef** names, see the IBM *IBM C/2 Fundamentals* book.)

errno.h

The **errno.h** include file defines the values that system-level calls use to set the *errno* variable. The **perror** routine uses the constants defined in **errno.h** to index the corresponding error message in the global variable *sys_errlist*.

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

fcntl.h

The **fcntl.h** include file defines flags used in the **open** and **sopen** calls to specify the type of the operations for which the file is open and to control whether the file is interpreted as a text file or a binary file. Include this file when you use **open** or **sopen**.

The routine declarations for **open** and **sopen** are not in **fcntl.h**. They are in the include file **io.h**.

float.h

The **float.h** include file contains definitions of constants that specify the ranges of floating-point data types, for example, the maximum number of digits for objects of type **double** (DBL_DIG = 15) or the minimum exponent for objects of type **float** (FLT_MIN_EXP = -38).

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

In addition, **float.h** defines floating-point-exception subcodes that the compiler uses with SIGFPE to trap floating-point errors. For additional information about SIGFPE, see the discussion of **signal.h** in this chapter.

io.h

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

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

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

limits.h

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

locking.h

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

The routine declaration for **locking** is in the **io.h** file. Use the **locking** routine only under DOS Version 3.30 or OS/2.

malloc.h

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

alloca	**_fmalloc**	**halloc**	**_msize**	**realloc**
calloc	**_fmsize**	**hfree**	**_nfree**	**sbrk**
_expand	**free**	**malloc**	**_nmalloc**	**stackavail**
_ffree	**_freect**	**_memavl**	**_nmsize**	

math.h

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

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

[1] **bessel** does not correspond to a single routine but to the routines named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**

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

The HUGE_VAL value, which is returned on error from some math routines, is defined in **math.h**. Use the HUGE_VAL value either as a manifest constant or as a global variable with **double** type. Do not change the value of the HUGE_VAL constant.

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

memory.h

The **memory.h** include file contains routine declarations for the seven buffer manipulation routines: **memccpy**, **memchr**, **memcmp**, **memcpy**, **memicmp**, **memset**, and **movedata**.

process.h

The **process.h** include file declares all process-control functions except for the **raise** and **signal** functions, declared in **signal.h**. The following functions are in the **process.h** file:

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

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

search.h

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

setjmp.h

The **setjmp.h** include file contains function declarations for the **setjmp** and **longjmp** functions. It also defines a system-dependent buffer type that the **setjmp** and **longjmp** functions use to save and restore the program state.

share.h

The **share.h** include file defines flags used in the **sopen** function to set the sharing mode of a file. Include this file whenever you use **sopen**. The function declaration for **sopen** is in the file **io.h**. Use the **sopen** function under DOS Version 3.30 or under OS/2.

signal.h

The **signal.h** include file defines the values for signals. The **raise** and **signal** functions are also declared in **signal.h**. DOS recognizes only the SIGINT (Ctrl+C) and the SIGFPE (floating-point exceptions) signals. OS/2 supports the additional signals SIGTERM, SIGUSR1, SIGUSR2, SIGUSR3, and SIGBREAK.

stat.h

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

stdarg.h

The **stdarg.h** include file defines macros that let you get access to arguments in functions with variable-length argument lists, such as **vprintf**. These macros are defined to be system-independent, portable, and compatible with the developing ANSI standard for C.

stddef.h

The **stddef.h** include file contains definitions of the commonly used pointers, variables, and types, from **typedef** statements, listed below:

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

stdio.h

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

clearerr	**fileno**[2]	**fseek**	**putchar**[2]	**sprintf**
fclose	**flushall**	**ftell**	**puts**	**sscanf**
fcloseall	**fopen**	**fwrite**	**putw**	**tempnam**
fdopen	**fprintf**	**getc**[2]	**remove**	**tmpfile**
feof[2]	**fputc**	**getchar**[2]	**rename**	**tmpnam**
ferror[2]	**fputchar**	**gets**	**rewind**	**ungetc**
fflush	**fputs**	**getw**	**rmtmp**	**vfprintf**
fgetc	**fread**	**printf**	**setbuf**	**vprintf**
fgetchar	**freopen**	**putc**[2]	**setvbuf**	**vsprintf**
fgets	**fscanf**	**perror**	**scanf**	

[2] Implemented as a macro.

The **stdio.h** file also defines the constants listed below. You can use these constants in your programs, but you should not alter their values.

BUFSIZ Buffers used in stream input/output must have a constant size, which is defined by the BUFSIZ constant. This value establishes the size of system-allocated buffers and must also be used when calling **setbuf** to allocate your own buffers.

_NFILE The _NFILE constant defines the number of open files allowed at one time. The five files, **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, remain open. You should include them when calculating the number of files your program opens. (The files **stdaux** and **stdprn** are not opened automatically under OS/2.)

EOF The EOF value is the value returned by an I/O routine when the end of the file (or in some cases, an error) is found.

NULL The NULL value is the null pointer value. It is 0 in small- and medium-model programs and 0L in large- and huge-model programs.

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

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

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

stdlib.h

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

abort	**ecvt**	**itoa**	**putenv**	**swab**
abs	**exit**	**labs**	**rand**	**system**
atof	**fcvt**	**ltoa**	**realloc**	**tolower**
atoi	**free**	**malloc**	**srand**	**toupper**
atol	**gcvt**	**onexit**	**strtod**	**ultoa**
bsearch	**getenv**	**perror**	**strtol**	**qsort**
calloc				

The **tolower** and **toupper** routines are functions in the run-time library, but the include file also uses them as macros in **ctype.h**. The declarations for **tolower** and **toupper** are enclosed in an **#ifndef** block; they take effect only if the compiler suppresses the corresponding macro definitions in **ctype.h** by removing the definitions of **tolower** and **toupper**.

The **stdlib.h** include file also includes the definition of the type **onexit_t** and declarations of the following global variables:

_doserrno	*_fmode*	*_osmode*	*sys_nerr*
environ	*_osmajor*	*_psp*	
errno	*_osminor*	*sys_errlist*	

string.h

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

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

timeb.h

The **timeb.h** include file (conventionally stored in a subdirectory named SYS) defines the **timeb** structure type and declares the **ftime** function, which uses the **timeb** structure type. Whenever you use the **ftime** function, you must include **timeb.h** to define the **timeb** structure type.

time.h

The **time.h** include file declares the time functions **asctime**, **ctime**, **difftime**, **gmtime**, **localtime**, **time**, and **tzset**. (The **ftime** and **utime** functions are declared in **sys\timeb.h** and **sys\utime.h**, respectively.)

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

types.h

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

utime.h

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

Chapter 5. Library Routines

abort

Purpose:

Stops the process that calls this function.

Format:

```
 /* Required for function declarations */
#include <stdlib.h>

void abort( )
```

Comments:

The **abort** function writes the message:

Abnormal program termination

to the **stderr** data stream, then calls raise (SIGABRT). The signal SIGABRT stops the process that called it, returning control to the process that initiated the calling process, usually the operating system. The **abort** function does not flush stream buffers or do **onexit** processing.

The **abort** function returns an exit status of 3 to the parent process or operating system.

Example:

The following example tests for successful opening of the file DATA and points to an error message if the opening of the file was unsuccessful.

```
#include <stdio.h>
#include <stdlib.h>
main()
{

FILE *stream;

if ((stream = fopen("data", "r")) == NULL)
   {
   perror("couldn't open data file");
   abort();
   }
}
```

Related Topics:

execl, execle, execlp, execv, execve, execvp, exit, _exit, raise, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, signal,

abs

Purpose:

Returns the absolute value of an integer argument.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

int abs(n)
int n;  /* Integer value */
```

Comments:

The **abs** function returns the absolute value of an integer argument *n*. There is no error return value. The result is undefined when the argument is the least of the negative **short int** (-32768), whose absolute value cannot be represented as a **short int**.

Example:

```
#include <stdlib.h>

int x = -4, y;

y = abs(x); /* y = 4 */
```

Related Topics:

cabs, fabs, labs

Purpose:

Tells whether you can get access to a specified file.

Format:

```
   /* Required for function declarations */
#include <io.h>

int access(pathname, mode)
char *pathname; /* File pathname */
int mode; /* Permission setting */
```

Comments:

The **access** function determines whether the specified file exists and whether you can get access to it in the given *mode*. The following list gives possible values for the *mode* and their meaning in the **access** call:

Value	Meaning
06	Check for permission to both read from and write to the file.
04	Check for permission to read from the file.
02	Check for permission to write to the file.
00	Check only for the existence of the file.

Because all existing files have read access under DOS, the modes 00 and 04 produce the same results on DOS. Similarly, the modes 06 and 02 produce the same results because, on DOS, getting access to write implies getting access to read.

The **access** function returns the value 0 if you can get access to the file in the given *mode*. A return value of -1 shows that the file does not exist or is not accessible in the given *mode*, and the system has set *errno* to one of the following values:

access

Value	Meaning
EACCES	Access is denied; the permission setting of the file does not allow you to get access to the file in the specified mode.
ENOENT	The system cannot find the file or or the path that you specified, or (in the OS/2 mode) the filename was incorrect.

Example:

The following example opens the file DATA for writing after checking the file to see if writing is permissible. The example prints the following error message:

data file not writeable: permission denied

if the permission setting prohibits you from writing to the file.

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

int fh;
main()
{
  /* Check for permission to write to the file */
if ((access("data",2)) == -1){
   perror("data file not writeable");
   exit(1);
   }

fh = open("data",O_WRONLY);
}
```

Related Topics:

chmod, fstat, open, stat

Purpose:

Returns the arc cosine of a value.

Format:

```
#include <math.h>

double acos(x)
double x;
```

Comments:

The **acos** function returns the arc cosine of *x* in the range 0 to π.

The value of *x* must be between -1 and 1. If *x* is less than -1 or greater than 1, **acos** sets *errno* to EDOM, writes a DOMAIN error message to the **stderr** data stream, and returns 0. For additional information about DOMAIN and EDOM, see "Math Errors" in Appendix A, "Error Messages," in this book.

You can change the way **acos** handles errors by using the **matherr** routine.

Example:

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

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

extern int errno;

main()
{
    float   x, y;

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

Related Topics:

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

Purpose:

Allocates space from the program's stack. The space is freed when the routine that called **alloca** is ended.

Format:

```
  /* Required for function declaration */
#include <malloc.h>

void *alloca(size)
  /* Bytes to be reserved from the stack */
size_t size;
```

Comments:

The **alloca** function returns a **char** pointer to the space reserved on the stack. The storage space to which the return value points is 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 **alloca** cannot reserve the space.

Example:

The following example shows a called function which allocates stack space to hold values having temporary existence. The goal is to compute the sum of the squares of random values drawn from a particular range, 0 to 1. The main program controls the number of integers to be processed. The called function performs the calculations and returns the sum of squares.

alloca

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

double sumsq(unsigned int);

main()
{
    unsigned int length = 10;

    printf("Sum of the squares of ");
    printf("%u values was ", length);
    printf("%e\n", sumsq(length));
}


double sumsq(length)
unsigned int length;
{
    float    *floatarray;
    double   d = 0.;
    unsigned int    i;

    /* allocate space on the stack for     */
    /* the float values      */
    floatarray = (float *) alloca(length *
    sizeof(float));

    /* fill the array with random numbers  */
    for (i = 0; i < length; i++)
    *(floatarray + i) = (float) rand()
       / 32768;
    for (i = 0; i < length; i++)
    d += *(floatarray + i) *
      *(floatarray + i);
    return(d);
}
```

Related Topics:

calloc, malloc, realloc

CAUTION:
Do not pass the pointer value returned by alloca as an argument to free. Also, because alloca manipulates the stack, use it only in simple assignment statements. Distinct stack-manipulation logic is used with every call; therefore, using alloca in an expression that is an argument to a function causes the stack pointer to be incorrect.

Purpose:

Converts time stored as a structure to a character string in storage.

Format:

```
#include <time.h>

char *asctime(time)
  /* Pointer to structure defined in TIME.H */
const struct tm *time;
```

Comments:

The **asctime** function converts *time* stored as a structure to a character string in storage. Obtain the *time* value from a call to **gmtime** or **localtime**, either of which returns a pointer to a **tm** structure, defined in **time.h**. See **gmtime** for a description of the **tm** structure fields.

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

```
Sat Jul 06 02:03:55 1985\n\0
```

The **asctime** function uses a 24-hour-clock format. All fields have constant widths. The newline character (\n) and the null character (\0) occupy the last two positions of the string.

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

asctime

Example:

This example polls the system clock and prints a message giving the current time.

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

struct tm *newtime;
time_t ltime;
main ()
{
  /* Get the time in seconds */
time(&ltime);
  /* Convert it to the structure tm */
newtime = localtime(&ltime);

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

Related Topics:

ctime, ftime, gmtime, localtime, time, tzset

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

Purpose:

Calculates the arc sine of a value.

Format:

#include <math.h>

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

Comments:

The **asin** function calculates the arc sine of *x* in the range $-\pi/2$ to $\pi/2$.

The value of *x* must be between -1 and 1. If *x* is less than -1 or greater than 1, **asin** sets *errno* to EDOM, writes a DOMAIN error message to **stderr** data stream, and returns a value of 0. For more information about EDOM and DOMAIN, see "Math Errors" in Appendix A, "Error Messages," in this book.

You can change the way the **asin** function handles errors by using the **matherr** routine.

asin

Example:

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

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

extern int errno;

main()
{
    float x, y;
    int j;
    char c;

    for (errno=EDOM; errno==EDOM; y=asin(x))
    {
  printf("Sine = ");
  j=scanf("%f", &x);
  if (j>0) errno=0;
  else scanf("%c", &c);
    }
    printf("Arcsine of %f = %f radians\n",x,y);
}
```

Related Topics:

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

Purpose:

Prints a diagnostic message and stops the process that called it if the *expression* is false.

Format:

#include <assert.h>

void assert(int *expression*)

Comments:

The **assert** routine prints a diagnostic message and stops the process that called it if the *expression* is false (zero). The diagnostic message has the following form:

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

Filename is the name of the source file and *linenumber* is the line number in the source file of the assertion that failed. The **assert** routine takes no action if the *expression* is true (nonzero).

Use the **assert** routine to identify program logic errors. Choose the *expression* so that it holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier, NDEBUG, to remove the **assert** calls from the program. If you define NDEBUG to any value with a **/D** command line option or with a **#define** directive, the C preprocessor removes all **assert** calls from the program source file.

There is no return value.

Note: The **assert** routine is a macro. You can use an **#undefine** directive to remove the **assert** macro definition to obtain access to an actual function called assert, which you supply.

assert

Example:

In this example, the assert routine tests the "string" argument for a null string and an empty string. Before processing this argument, it verifies that the "length" argument is positive.

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

void analyze(char *, int);

main()
{
    analyze("",1);
}

void analyze(string, length)
char *string;
int length;
{
 assert(length > 0);
 assert(string != NULL);
 assert(*string != '\0');
}
```

Purpose:

Calculates the arc tangent of *x* or *y/x*.

Format:

```
#include <math.h>

/* Calculate arc tangent of x*/
double atan(x)
double x;

/* Calculate arc tangent of y/x */
double atan2(y, x);
double x;
double y;
```

Comments:

The **atan** and **atan2** functions calculate the arc tangent of *x* and *y/x*, respectively.

The **atan** function returns a value in the range $-\pi/2$ to $\pi/2$; the **atan2** function returns a value in the range $-\pi$ to π. If both arguments of **atan2** are zero, the function sets *errno* to EDOM, writes a DOMAIN error message to the **stderr** data stream, and returns a value of 0.

You can change the way these routines handle errors by using the **matherr** routine.

Example:

This example is a function that converts Cartesian to polar coordinates.

```
#include <math.h>
struct polar {
   double r;     /* Radius */
   double theta; /* Angle in radians */
   };
struct polar *cart_polar(f)
struct complex *f;
{
   struct polar *p;
   p->r = cabs(*f);
   p->theta = atan2(f->y,f->x);
   return (p);
}
```

Related Topics:

acos, asin, cos, matherr, sin, tan

Purpose:

Converts character strings to **double**, **integer**, or **long**.

Format:

```
   /* Required for function declarations */
#include <stdlib.h>

   /* Convert string to double */
double atof(string)

   /* Convert string to int */
int atoi(string)

   /* Convert string to long */
long int atol(string)
const char *string;  /* String to be converted */
```

Comments:

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

The **atof** function expects a *string* in the following form:

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

The *whitespace* consists of required space and tab characters, which the function ignores. The *sign* is either "+" or "-". The *digits* are one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter "d", "D", "e", or "E". The exponent is a decimal integer, which may be signed.

atof - atol

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**.

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

Examples:

This program shows how numbers stored as strings can be converted to numerical values.

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

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

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

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

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

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

}
```

Related Topics:

ecvt, fcvt, gcvt

bdos

Purpose:

Makes a DOS system call.

Format:

#include <dos.h>
int bdos(*dosfn*, *dosdx*, *dosal*)
int *dosfn*; /* Function number */
unsigned int *dosdx*; /* DX register value */
unsigned int *dosal*; /* AL register value */

Comments:

The **bdos** function makes the DOS system call specified by *dosfn* after placing the value specified by *dosdx* in theDXregister and the value specified by *dosal* in the AL register. The **bdos** function performs an INT 21H instruction that makes the system call. When control returns from DOS, **bdos** returns the contents of the AX register.

Use the **bdos** function to make DOS system calls that either take no arguments or only take arguments stored in the DX (DH,DL) and/or AL registers.

The **bdos** function returns the value of theAX register only after DOS completes the system call.

Example:

This example makes DOS function call 9 (display string) display a prompt. Because this call does not need the AL register value, code a 0 instead. This example works correctly only in small and medium model programs.

```
#include <dos.h>

char *buffer = "Enter file name:$";

   /* AL is not needed, so 0 is used */
bdos(9,(unsigned)buffer,0);
```

Related Topics:

intdos, intdosx

Notes:

1. Do *not* use this call to make system calls that indicate errors by setting the carry flag. Because C programs do not have access to this flag, they cannot tell the status of the return value. Use the **intdos** function in these cases.
2. The **bdos** function is not available under OS/2. For information on calling OS/2 functions from a C program, see "Application Program Interface" in the *IBM Operating System/2 Technical Reference* manual.

bessel

Purpose:

Returns bessel functions.

Format:

```
#include <math.h>
double j0(x)
double j1(x)
double jn(n,x)
double y0(x)
double y1(x)
double yn(n,x)
double x;  /* Floating-point value */
int n;  /* Integer order */
```

Comments:

Bessel functions are power-series expansions that are solutions of certain special differential equations. These equations occur in problems in physics, particularly those problems that concern oscillations.

The **j0**, **j1**, and **jn** routines are bessel functions of the first kind for orders 0, 1, and *n*, respectively.

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

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

For **j0**, **j1**, **y0**, or **y1**, if *x* is too large, the function sets *errno* to ERANGE, writes a TLOSS error message to the **stderr** data stream, and returns 0.

For **y0**, **y1**, or **yn**, if *x* is negative, the function sets *errno* to EDOM, writes a DOMAIN error message to the **stderr** data stream, and returns the value HUGE_VAL.

For more information about EDOM and DOMAIN, see "Math Errors" in Appendix A, "Error Messages" in this book.

You can change the way that the **bessel** function handles errors by using the **matherr** routine.

Example:

```
#include <math.h>
#define  j2(x)     (2./(x)) * j1(x) - j0(x)

/* The Bessel functions of the first
kind obey the recurrence relation:

   J(n+1,x) = (2n/x)*J(n,x) - J(n-1,x)  */
```

This implies that the particular function jn(2,x) may be expressed in terms of j0 and j1 by the identity given above in the #define statement. This example tests whether the library function jn(2,x) computes values identical to those of the macro j2, using selected double values of x.

```
double g[3]={ 6.E-2, 6.25E-2, 6.5E-2 };
main()
{
    int i;
    double d, e, f;

printf("   x             j2(x)        "
     "jn(2,x)             compare?\n");
    for(i=0; i<3; i++)
    {
         d=g[i];
         e=j2(d);
         f=jn(2,d);
         printf("%7.4f   %16.13e"
             "  %16.13e     ",d,e,f);
         printf((e==f)?"  equal\n"
                 :"unequal\n");     }
}
```

Related Topics:

matherr

bsearch

Purpose:

Performs a binary search of a sorted array.

Format:

```
  /* Use either search.h or stdlib.h */
#include <stdlib.h>

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

   /* Pointer to base of search data */
const void *base;
   /* Number and width of elements */
size_t num, width ;
   /* Pointer to compare function */
int (*compare)( const void *element1,const void *element2) ;
```

Comments:

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

Compare is a pointer to a routine, which you must supply, that compares two array elements and returns a value specifying their relationship. The **bsearch** function calls this routine one or more times during the search, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values.

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

The **bsearch** function returns a pointer to the first occurrence of the *key* in the array to which the *base* points. If **bsearch** cannot find the *key*, it returns NULL.

Example:

This program reads the command-line parameters and uses **qsort** to sort them. It then displays the sorted arguments. Next, it uses **bsearch** to locate the first parameter starting with "TEMP".

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

int qcompare(); /* function for qsort */
int bcompare(); /* function for bsearch */

main(argc, argv)
int argc;
char **argv;
{
    char **result;
    char *key = "TEMP";
    int i;

/* Eliminate argv[0] from sort:    */
    argv++;
    argc--;

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

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

/* Find item that begins with
 * "TEMP" using binary search.     */
    result=(char **)bsearch((char *)&key,
           (char *)argv, argc,
           sizeof(char *), bcompare);
    if(result)
         printf("%s found\n", *result);
    else
         printf("TEMP not found!\n");
}
```

bsearch

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

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

Related Topics:

qsort

Purpose:

Calculates the absolute value of a complex number.

Format:

```
#include <math.h>
double cabs(z)
  /* Contains real and imaginary parts */
struct complex z;
```

Comments:

The **cabs** function calculates the absolute value of a complex number. The complex number must be a structure with type **complex**, defined in **math.h**:

struct complex {double *x,y*;**};**

A call to **cabs** is equivalent to

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

The **cabs** function returns the absolute value as described above. If an overflow results, the function calls the **matherr** routine, sets *errno* to ERANGE and returns the value HUGE_VAL.

cabs

Example:

The following example computes *d* to be the the absolute value of the complex number (3.0, 4.0).

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

main()
{
    struct complex value;
    double d;

    value.x = 3.0;
    value.y = 4.0;

    d = cabs(value);
    printf("Absolute value is %f\n",d);
}
```

Related Topics:

abs, fabs, hypot, labs

Purpose:

Reserves storage space for an array and sets the initial value of all elements to 0.

Format:

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

Comments:

The **calloc** function reserves storage space for an array of *n* elements, each of length *size* bytes. The **calloc** function then gives each element an initial value of 0.

The **calloc** function returns a pointer to the reserved space. The storage space to which the return value points is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage available.

calloc

Example:

The following example reserves enough space in storage for 40 long integers and gives each integer an initial value of zero.

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

long *lalloc;
main()
{
lalloc = (long *)calloc(40,sizeof(long));
If (lalloc !=NULL)
   printf("Allocation OK \n");
else printf ("calloc failed \n");
}
```

Related Topics:

free, malloc, realloc

Purpose:

Returns a double value representing the smallest integer that is greater than or equal to x.

Format:

#include <math.h>

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

Comments:

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

There is no error return value.

Example:

The following example sets the *y* to the smallest integer greater than 1.05 and, then, the smallest integer greater than -1.05. The results are 2. and -1., respectively.

```
#include <math.h>

double y, z;

        .
        .
        .
y = ceil(1.05);   /* y = 2.0 */
z = ceil(-1.05);  /* z = -1.0 */
```

Related Topics:

floor, fmod

cgets

Purpose:

Reads and stores a string of characters directly from the keyboard.

Format:

```
   /* Required for function declarations */
#include <conio.h>

char *cgets(str)
   /* Storage location for data */
char *str;
```

Comments:

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

The **cgets** function continues to read characters until it meets a carriage return/line feed combination or reads the specified number of characters. It stores the string starting at *str*[2]. If **cgets** reads a CR-LF combination, it replaces this combination with a null character (\0) before storing the string. The **cgets** function then stores the actual length of the string in the second array element, *str*[1].

The **cgets** function returns a pointer to the start of the string, which is at *str*[2]. In case of error in OS/2 mode, **cgets** returns NULL.

Example:

This example 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>
char buf[82];
char *result;

main()
{
    buf[0] = 80;  /* max. number */
    printf("Input line of text, follo");
    printf("wed by carriage return:\n");
    result = cgets(buf);
    printf("\nLine length = %d\n",buf[1]);
    printf("Text = %s\n",result);
}
```

Related Topics:

getch, getche

chdir

Purpose:

Changes the directory.

Format:

```
   /* Required for function declarations */
#include <direct.h>

int chdir(pathname)
   /* Path name of new working directory */
char *pathname;
```

Comments:

The **chdir** function causes the current working directory to change to the directory specified by *pathname*. The *pathname* must refer to an existing directory.

The **chdir** function returns a value of 0 if the working directory successfully changes. A return value of -1 shows an error; in this case, **chdir** sets *errno* to ENOENT, showing that **chdir** cannot find the specified pathname. No error occurs if *pathname* specifies the current working directory.

Example:

The following example changes the current working directory to the root directory.

```
#include <direct.h>

chdir("\\");
```

Related Topics:

mkdir, rmdir, system

Note: This function can change the current working directory only on the current default drive. It cannot change the current working directory on a different drive. For example, if A:\BIN is the current working directory, the following does not change it:

```
chdir ("c:emp");
```

Under DOS you can achieve the desired result. In this case, you must first call **system** to change the current default drive to C: before you can change the current working directory on that drive.

chmod

Purpose:

Changes the permission setting of a file.

Format:

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

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

Comments:

The **chmod** function changes the permission setting of the file specified by *pathname*. The permission setting controls access to the file for reading or writing. The *pmode* constant expression contains one or both of the manifest constants S_IWRITE and S_IREAD, defined in **sys\stat.h**. The **chmod** functions ignores any other values for *pmode*. When you give both constants, the **chmod** function joins them with the bitwise operator OR(|). The following list gives the meaning for the values of the *pmode* argument.

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing both permitted.

If you do not give permission to write to the file, the **chmod** function makes the file read-only. Under DOS, all files are readable; it is not possible to give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE set the same permission.

The **chmod** function returns the value 0 if it successfully changes the permission setting. A return value of -1 shows an error; in this case, the **chmod** function sets *errno* to ENOENT, showing either that it cannot find the specified file or in the OS/2 mode the filename was incorrect.

Example:

This program takes file names passed as arguments and sets each to read-only.

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

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

{
int result;
register char **p;
if (argc<2) return;
for (p=++argv; argc>1; argc--, p++) {
      /* Make a file read-only */
    result = chmod(*p, S_IREAD);
    if (result == -1)
    perror("file not found");
    }
}
```

Related Topics:

access, creat, fstat, open, stat

chsize

Purpose:

Lengthens or cuts off the file associated with the *handle*.

Format:

```
 /* Required for function declarations */
#include <io.h>

int chsize(handle, size)
 /* A handle referring to open file */
int handle;
 /* The new length of the file, in bytes */
long size;
```

Comments:

The **chsize** function lengthens or cuts off the file associated with the *handle* to the length specified by *size*. You must open the file in a mode that permits writing. The **chsize** function adds null characters (\0) when it lengthens the file. When **chsize** cuts off the file, it erases all data from the end of the shortened file to the end of the original file.

The **chsize** function returns the value 0 if it successfully changes the file size. A return value of -1 shows an error, and **chsize** sets *errno* to one of the following values.

Value	Meaning
EACCESS	The specified file is locked against access
EBADF	The file handle is not valid, or the file is not open for writing.
ENOSPC	There is no space left on device.

Example:

This program opens a file named **dat** and writes data to it. Then it uses **chsize** to extend the size of **dat**.

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

#define MAXSIZE 32768L

int fh, result;
char buffer[BUFSIZ] = "Initialize the buffer to
        this string.\n";
main ()
{
int i;
unsigned int nbytes = BUFSIZ;
fh = open("dat",O_RDWR|O_CREAT,
S_IREAD|S_IWRITE);
for (i=0; i<50; i++)
result = write(fh, buffer, nbytes);
result = -1;

  /* Make sure the file is no longer */
  /* than 32K bytes before closing it */
if (lseek(fh,0L,SEEK_END) > MAXSIZE)
   result = chsize(fh,MAXSIZE);
if (result == 0)
   printf ("Size successfully changed");
else
   printf ("Problem in changing the size");
}
```

Related Topics:

close, creat, open

_clear87

Purpose:

Gets and clears the floating-point status word.

Format:

```
#include <float.h>
unsigned int _clear87( )
```

Comments:

The **_clear87** function gets and clears the floating-point status word. The floating-point status word is a combination of the numeric coprocessor status word and other conditions that the numeric exception handler detects, such as floating-point stack overflow and underflow.

The bits in the value returned show the floating-point status. For a complete definition of the bits returned by **_clear87**, see the **float.h** include file.

Example:

The following example shows how you can lose significance by assigning a **double** to a **float**. It takes a number close to zero as a **double** and assigns it to a **float**. The result is a loss of significance and the creation of a denormal number. The **_clear87** function gets the floating-point status word, and the **printf** function prints it as immediate data.

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

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

main()
   {
   unsigned int statword;

   statword = _clear87();
   printf("cleared floating-point status word"
          " = %.4x\n", statword);

/* Assignment of the double to the float y
   is inexact; the underflow bit is also set. */
   y=a;
   statword = _clear87();
   printf("floating-point status = %.4x"
          "after underflow\n", statword);

/* Reassigning the denormal y to the double b
   causes the denormal bit to be set.          */
   b = y;
   statword = _clear87();
   printf("floating-point status = %.4x"
          "for denormal\n",statword);
}
```

Output:

```
cleared floating-point status word = 0000
floating-point status = 0030 after underflow
floating-point status= 0002 for denormal
```

Related Topics:

_control87, _status87

clearerr

Purpose:

Resets the error and end-of-file indicators.

Format:

#include <stdio.h>
void clearerr (*stream*)
FILE **stream*;/* Pointer to file structure */

Comments:

The **clearerr** function resets the error indicator and end-of-file indicator for the specified *stream* to 0. The system does not automatically clear error indicators. When **clearerr** sets the error indicator for a specified stream, operations on that stream continue to return an error until your program calls **clearerr** or **rewind**.

Example:

The following example sends data to a stream, and then checks to make sure that a write error has not occurred. The stream must be open for writing.

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

FILE *stream;
int c;
main()
{
stream = fopen("data", "w");
if ((c=getc(stream)) == EOF) {
   if (ferror(stream)) {
      fprintf(stderr,"write error\n");
      clearerr(stream);
      }
   }
}
```

Related Topics:

eof, feof, ferror, perror

close

Purpose:

Closes the file associated with the *handle*.

Format:

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

Comments:

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

The **close** function returns 0 if it successfully closes the file. A return value of -1 shows an error, and **close** sets *errno* to **EBADF**, showing a incorrect file handle argument.

Example:

The following example closes the file DATA after opening it as a read-only file with the file handle *fh*.

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

int fh;

fh = open("data",O_RDONLY);
   .
   .
   .
close(fh);
```

Related Topics:

chsize, creat, dup, dup2, open, unlink

Purpose:

Gets and sets the floating-point control word.

Format:

```
#include <float.h>

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

Comments:

The **_control87** function gets and sets the floating-point control word. The floating-point control word lets the program change the precision, rounding, and infinity modes in the floating-point math package. You can mask or unmask floating-point exceptions using the **_control87** function.

If the value for the mask is equal to 0, **_control87** gets the floating-point control word. If the mask is non-zero, **_control87** sets a new value for the control word in the following manner. For any bit in the mask equal to 1, the corresponding bit in *new* updates the control word. This is equivalent to the expression

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

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

The bits in the returned value show the floating-point control state. For a complete definition of the bits returned by **_control87**, see the discussion of the **float.h** include file.

_control87

Example:

This example prints the control word in hexadecimal, then illustrates different representations of 0.01, depending on the precision.

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

double a = .1;

main()
   {
   printf("control = %.4x\n",
      /* Get control word */
   _control87(CW_DEFAULT,0));
   printf("a*a = .01 = %.15e\n",a*a);
  _control87(PC_24,MCW_PC);
 /* Set precision to 24 bits */
   printf("a*a =.01 (rounded to 24 bits) =%.15e\n",a*a);
       /* Restore to initial default */
   _control87(CW_DEFAULT,0xffff);
   printf("a*a = .01 = %.15e\n",a*a);
   }
```

Related Topics:

_clear87, _status87

Purpose:

The **cos** function returns the cosine. The **cosh** function returns the hyperbolic cosine.

Format:

```
#include <math.h>

  /* Calculate the cosine of x */

double cos(x)

  /* Calculate the hyperbolic cosine of x */
double cosh(x)
double x;  /* Angle in radians */
```

Comments:

The **cos** function returns the cosine of *x*. If *x* is large, a partial loss of significance in the result might occur. In such cases, **cos** produces a PLOSS error, but prints no message. If *x* is so large that a total loss of significance results, **cos** prints a TLOSS error message to STDERR and returns 0 In both cases, the **cos** function sets *errno* to ERANGE.

The **cosh** function returns the hyperbolic cosine of *x*. If the result is too large, **cosh** returns the value HUGE_VAL and sets *errno* to ERANGE. You can changes the way that **cosh** handles errors by using the **matherr** routine.

For more information about PLOSS, TLOSS, ERANGE, and HUGE_VAL, see "Math Errors" in Appendix A, "Error Messages."

Example:

This program samples the level of an oscillator at ten equally-spaced points, one millisecond apart. The phase angle begins at zero, at time zero.

```
#include <math.h>
#define  TWOPI    6.283185307
 /*        frequency in hertz  */
double frequency=120.0;
double amplitude=1.0;

main()
{
    double level, time;
    int n;

    printf("time (s)     level\n");
    for (n=0; n<10; n++)
    {
  time=n * 0.001;

 level =
 amplitude*cos(TWOPI*frequency*time);
  printf(" %5.3f     % 8.6f\n",
   time,level);
    }
}
```

Related Topics:

acos, asin, atan, atan2, matherr, sin, sinh, tan, tanh

Purpose:

Formats and prints characters directly to the screen.

Format:

```
   /* Required for function declarations */
#include <conio.h>

int cprintf(format-string[,argument...])
   /* Format control string */
char *format-string;
```

Comments:

The **cprintf** function formats and prints a series of characters and values directly to the screen, using the **putch** function to put each character out to the screen. The **cprintf** function converts each *argument* (if any) and puts it out according to the corresponding format specification in the *format-string*. The *format-string* has the same form and function as the *format-string* argument for the **printf** function. See the **printf** reference page for a description of the *format-string* and arguments.

The **cprintf** function returns the number of characters printed.

Example:

The following example prints:

i=-16, j=0x1d, k=511

```
#include <conio.h>

int i = -16, j = 29;
unsigned int k = 511;
main()
{
cprintf("i=%d, j=%#x, k=%u\n", i, j, k);
}
```

cprintf

Related Topics:

fprintf, printf, sprintf

Note: Unlike the **fprintf**, **printf**, and **sprintf** functions, **cprintf** does not translate line feed characters into output of carriage return/line feed combinations.

Purpose:

Writes a string ending with a null character directly to the screen.

Format:

```
  /* Required for function declarations */
#include <conio.h>

int cputs(str)
char *str;  /* Pointer to output string */
```

Comments:

The **cputs** function writes directly to the screen the string to which **str* points. The string *str* must end with a null character (\0). The **cputs** function does not automatically add a carriage return/line feed combination to the string after writing.

If the action is successful, **cputs** returns 0. In case of error in OS/2, **cputs** returns EOF.

Example:

The following statement puts a prompt out to the screen.

```
#include <conio.h>

char *buffer = "Insert data disk in drive a: \r\n";

cputs(buffer);
```

Related Topics:

putch

creat

Purpose:

Creates or opens and cuts off a file.

Format:

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

int creat(pathname, pmode)
char *pathname; /* Pathname of new file */
int  pmode; /* Permission setting */
```

Comments:

The **creat** function either creates a new file or opens and cuts off an existing file. If the file specified by *pathname* does not exist, **creat** creates a new file is with the given permission setting and open for writing. If the file already exists and its permission setting allows writing, **creat** cuts off 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 you close it for the first time. The *pmode* integer expression contains one or both of the manifest constants S_IWRITE and S_IREAD, defined in **sys\stat.h**. When you give both constants, **creat** joins them with the bitwise operator OR(|).

The following gives the values of the *pmode* argument and their meaning.

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If you do not give permission to write to the file, the file is a read-only file. Under DOS it is not possible to give write-only permission. Thus, the modes S_IWRITE and S_IREAD |S_IWRITE have the same results. Under DOS, files opened using **creat** are always open in DOS mode (see **sopen** in this chapter). The **creat** function applies the current file permission mask to *pmode* before setting the permissions (see **umask** in this chapter). The **creat** function returns a handle for the created file if the call is successful. A return value of -1 shows an error, and **creat** sets *errno* to one of the following values:

Value	Meaning
EACCCES	The pathname specifies an existing read-only file or specifies a directory instead of a file.
EMFILE	No more file handles are available. There are too many open files.
ENOENT	The pathname was not found, or (in OS/2 mode) the filename was incorrect.

Example:

This example creates for reading or writing a file with the file name DATA. It prints an error message if the **creat** operation fails.

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
main()
{
    int fh;

    fh=creat("data",S_IREAD|S_IWRITE);
    if(fh==-1)
        perror("couldn't create data file");
    else printf("file \"data\" created\n"); }
```

creat

Related Topics:

chmod, chsize, close, dup, dup2, open, sopen, umask

Note: IBM provides the **creat** function primarily for compatibility with previous libraries. A call to **open** with the O_CREAT and O_TRUNC values specified in the *oflag* argument has the same results as **creat** and is preferable for new code.

Purpose:

Reads data directly from the keyboard to locations given by *arguments*.

Format:

```
  /* Required for function declarations */

#include <conio.h>

int cscanf(format-string[,argument...])
char *format-string; /* Format control string */
```

Comments:

The **cscanf** function reads data directly from the keyboard to the locations given by the *arguments*, if any. The **cscanf** function uses the **getche** function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function. See the **scanf** reference page for a description of the *format-string*.

The **cscanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

Example:

The following example stores string input from the keyboard. The result is the number of correctly matched input fields. If **cscanf** can match no input fields, the result is zero.

```
#include <conio.h>

int result;
char buffer[20];

        .
        .
        .
cprintf("Please enter file name:");

result = cscanf("%19s", buffer);
```

Related Topics:

fscanf, scanf, sscanf

Note: Although **cscanf** normally echoes the input character, it will not do so if the last action was an **ungetch**.

Purpose:

Converts time stored as **long** value to a character string.

Format:

```
  /* Required only for function declarations */
#include <time.h>

char *ctime(time)
const time_t *time;   /* Pointer to stored time */
```

Comments:

The **ctime** function usually obtains the time value 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:

```
Sat Jul 06 02:03:55 1985\n\0
```

The **ctime** function uses a 24-hour clock format. All fields have a constant width. The newline character (\n) and the null character (\0) occupy the last two positions of the string.

Under DOS, **ctime** does not understand dates prior to 1980. If *time* represents a date before January 1, 1980, **ctime** returns NULL.

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

ctime

Example:

This example gets the number of seconds elapsed since January 1, 1970 00:00:00 GMT and assigns it to ltime. It then uses the **ctime** function to convert the number of seconds to the current time. It prints a message giving the current date and time.

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

main()
{
    time_t ltime;

    time(&ltime);
    printf("The time is %s\n",
       ctime(&ltime));
}
```

Related Topics:

asctime, ftime, gmtime, localtime, time

Note: The **asctime** and **ctime** functions use a single, statically-allocated buffer for holding the return string. Each call to one of these functions destroys the result of the previous call.

Purpose:

The **cwait** function delays the completion of a parent process until a particular child process is complete.

Format:

#include <process.h>

int cwait (*stat_loc,process_id,action_code*)
int **stat_loc*;
int *process_id*;
int *action_code*;

Comments:

The **cwait** function delays a parent process until the child process specified by *process_id* ends. You can use this function only under OS/2.

The *process_id* specifies the child process for which the parent process waits. This value is the *process_id* value returned by the **spawn** function or by the call to the OS/2 DOSEXECPGM function that started the child process. If the specified child process ends before **cwait** is called, **cwait** returns immediately. If the *process_id* is 0, the parent process waits until all of its child processes end.

If not NULL, the *stat_loc* argument points to a location that holds information about the return status and the return code of the child process. The return status shows whether the child process ended normally, using a call to the OS/2 DOSEXIT function. The low-order and high-order bytes of the return status are as follows:

Byte	Contents
Low-order	0
High-order	The low-order byte of the resulting code that the child process passed to DOSEXIT. DOSEXIT 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 if control from the child process fell through at the end of **main**, an unpredictable value.

If the child process stopped for any other reason, the low-order and high-order bytes of the return status are as follows:

Byte	Contents
Low-order	Return code from OS/2 DOSCWAIT function.

Code	Meaning
1	Hard-error abnormal end
2	Trap operation
3	SIGTERM signal not intercepted

High-order	0

The action-code specifies when the parent process starts running again as shown in the following list:

Action Code Meaning

WAIT_CHILD The parent process waits until the specified child process stops.

WAIT_GRANDCHILD
The parent process waits until the specified child process and all of the child processes of that child process stop.

WAIT_CHILD and WAIT_GRANDCHILD are defined in **process.h**.

If **cwait** returns after an unexpected end of the child process, it returns -1 to the parent process and sets *errno* to EINTR.

If **cwait** returns after a normal end of the child process, it returns the process identifier of the child process to the parent process.

Otherwise, **cwait** returns immediately with a value of -1. In this case, *errno* indicates the error, as shown in the following list:

Value	Meaning
EINVAL	Incorrect action code
ECHILD	No child process exists, or incorrect process identifier.

Related Topics:

exit, _exit, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, wait

dieeetomsbin - dmsbintoieee

Purpose:

Converts double-precision numbers between binary format and IEEE format.

Format:

```
#include <math.h>

    /* IEEE double precision to binary double */
int dieeetomsbin(src8,dst8)

    /* binary double to IEEE double precision */
int dmsbintoieee(src8,dst8)
double *src8, *dst8;
```

Comments:

The **dieeetomsbin** routine converts a double-precision number in IEEE format to binary format. The **dmsbintoieee** routine converts a double-precision number in binary format to IEEE format.

These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

Example:

This example uses a particular value (7.) to test whether these two routines function as inverses of each other. After conversion and reconversion, the value (7.) is written out.

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

main()
{
   double  b=7.;
   double  c, d;

   if (dmsbintoieee(&b, &c) == 1)
           fprintf(stderr,
"overflow converting to IEEE form\n");
   if (dieeetomsbin(&c, &d) == 1)
           fprintf(stderr,
"overflow converting from IEEE form\n");

   printf("The number after"
          "reconversion is %f\n",d);
}
```

Related Topics

fieeetomsbin, fmsbintoieee

Note: These routines do not handle IEEE NaNs and infinities. IEEE denormals are treated as 0 in the conversions.

difftime

Purpose:

Computes the difference between *time2* and *time1*.

Format:

```
#include <time.h>

double difftime(time2,time1)
    /* Type time_t defined in time.h */
time_t time2;
time_t time1;
```

Comments:

The **difftime** function computes the difference between *time2* and *time1*. **Difftime** is a macro.

The **difftime** function returns the elapsed time in seconds from *time1* to *time2* as a double precision number.

Example:

The following example shows a timing application using **difftime**. The example calculates how long it takes to find the prime numbers from 3 to 10000.

```
#include <time.h>

int mark[10000];

main()
{
    time_t start, finish;
    register int i, loop, n, num, step;

    printf("This program will take about 3 minutes"
         "on an AT or 8 minutes on a PC.\n");
    printf("Working...\n");

    time(&start);
    for (loop = 0; loop < 1000; ++loop)
        for (num = 0, n = 3; n < 10000; n += 2)
            if (!mark[n])
            {
                step = 2*n;
                for(i= 3*n; i<10000; i += step)
                     mark[i] = -1;
                ++num;
            }
    time(&finish);

  /*   Divide elapsed time by 1000 to get the
   *   average time per pass through the
   *   loop.                                  */
    printf("\nProgram takes %.3f seconds "
           "to find %d primes.\n",
           difftime(finish,start)/1000.,num);
}
```

The program takes an average of 0.110 seconds to find each of the 1228 primes.

Related Topics:

time

dosexterr

Purpose:

Obtains and stores values from DOS system call function 59H.

Format:

#include <dos.h>

int dosexterr (*buffer*)
struct DOSERROR **buffer*;

Comments:

The **dosexterr** function obtains the register values returned by the DOS system call 59H and stores the values in the structure to which *buffer* points. Use this function when you make system calls under DOS, which offers extended error handling. See your *IBM Disk Operating System Technical Reference* book for details about DOS system calls.

The **dos.h** include file defines the structure type DOSERROR as follows:

```
struct DOSERROR {
  int exterror;
  char class;
  char action;
  char locus;
  };
```

Giving a null pointer argument causes **dosexterr** to return the value in AX without filling in the structure fields.

The **dosexterr** function returns the value in the AX register (identical to the value in the **exterror** structure field). This code identifies which particular error condition arose. The **class** field of the structure gives the type of error, such as permission, hardware, or media failure. The **action** field recommends how the user can remedy the problem. The **locus** field gives additional information to locate the failure, such as network, memory, or block device. Codes for all of these returns are in *IBM Disk Operating System Technical Reference*.

Example:

This example tries to open a nonexistent file called VOID for reading. When the file fails to open, the example prints extended error codes for class, action, and locus.

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

struct DOSERROR doserror;
int fd;

main()
{
    if((fd=open("void",O_RDONLY))==-1)
    {
         dosexterr(&doserror);
         printf("error=%d, class=%d,"
              "action=%d, locus=%d\n",
     doserror.exterror,doserror.class,
     doserror.action,doserror.locus);
    }
}
```

Related Topics:

perror

Note: The **dosexterr** function should be used only under DOS Version 3.30. This routine is not supported under OS/2.

dup - dup2

Purpose:

Associates a second file handle with a currently open file.

Format:

```
  /* Required for function declarations */
#include <io.h>

  /* Create second handle for open file */
int dup(handle)
  /* Handle referring to open file */
int handle;

  /* Force handle2 to refer to handle file */
int dup2(handle1, handle2)
int handle1; /* Handle referring to open file */
int handle2; /* Any handle value */
```

Comments:

The **dup** and **dup2** functions associate a second file handle with a currently-open file. You can carry out operations on the file, using either file handle, because all handles associated with a given file use the same file pointer. Creation of a new handle does not affect the type of access allowed for the file. The **dup** function returns the next available file handle for the given file. The **dup2** function forces the given handle, *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.

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	This is an incorrect file handle.
EMFILE	No more file handles are available. There are too many open files.

Example:

The following example makes another file handle refer to the same file as file handle 1 (**stdout**). Then, it makes file handle 3 refer to the same file as file handle 1 (**stdout**). If file handle 3 is already open, this example first closes the file.

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

int fh;

     .
     .
     .

fh = dup(1);

if (fh == -1)
   perror("dup(1) failure");

fh = dup2(1,3);

if (fh != 0)
   perror("dup2(1,3) failure");
```

Related Topics:

close, creat, open

ecvt

Purpose:

Converts a floating-point number to a character string.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *ecvt(value, ndigits, decptr, signptr)
double value; /* Number to be converted */
int ndigits; /* Number of digits stored */
  /* Pointer to stored decimal position */
int *decptr;
  /* Pointer to stored sign indicator */
int *signptr;
```

Comments:

The **ecvt** function converts a floating-point number to a character string. The *value* is the floating-point number to be converted. The **ecvt** function stores *ndigits* digits of *value* as a string and adds a null character (\0). If the number of digits in *value* exceeds *ndigits*, the low-order digit is rounded. If there are fewer than *ndigits* digits, the string is padded with zeros.

Only digits are stored in the string. You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr*. The *decptr* points to an integer value that gives the position of the decimal point with respect to the beginning of the string. A 0 or a negative integer value indicates that the decimal point lies to the left of the first digit. The *signptr* points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

The **ecvt** function returns a pointer to the string of digits. There is no error return value. Because of the linited precision of the double type, no more than 16 decimal deigits are significant in any conversion.

Example:

This example will read in two floating-point numbers, compute their product, and print out only the billions digit of its character representation. At most 16 decimal digits of significance can be expected.

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

main()
{
    float x, y;
    double z;
    int w, b, i, decimal, sign;
    char c;
    char *buffer;

    printf("Enter two "
         "floating-point numbers.\n");
          if(2!=(i=scanf("%e %e",&x,&y)))
    {
         printf("input error...\n");
         abort();
    }
    z=x*y;
    w=log10(fabs(z))+1.;
    buffer=ecvt(z,w,&decimal,&sign);
    b=decimal-10;
    if(b<0)
         printf("Their product does not"
              " exceed one billion.\n");
    if(b>15)
         printf("The billions digit of"
 " their product is insignificant.\n");
    if((b>-1)&&(b<16))
         printf("The billions digit of"
              "their product is %c.\n",
              buffer[b]);
}
```

Related Topics:

atof, atoi, atol, fcvt, gcvt

Note: The **ecvt** and **fcvt** functions use a single, statically-allocated buffer for the conversion. Each call to one of these functions destroys the result of the previous call.

eof

Purpose:

Determines the position of the end-of-file character.

Format:

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

Comments:

The **eof** function determines whether the file pointer has reached the end-of-file character for the file associated with *handle*.

The **eof** function returns the value 1, if the current position is the end of the file, or 0, if it is not. A return value of -1 shows an error, in which case the system sets *errno* to EBADF, showing an incorrect file handle.

Example:

This example counts the number of bytes that it reads in a file until the EOF symbol.

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

main()
{
    int fh;
    unsigned count,total=0;
    char buf[5];

    fh=open("data",O_RDONLY);

    while(!eof(fh))
    {
         count=read(fh,buf,1);
         total++;
    }
    printf("%u bytes were read.\n",total);

}
```

Related Topics:

clearerr, feof, ferror, perror

execl - execvp

Purpose:

Load and run a new child process.

Format:

```
  /* Required for function declarations */
#include <process.h>

int execl(pathname, arg0, arg1,...,
        argn,NULL)
int execle(pathname, arg0, arg1,...
        argn, NULL, envp)
int execlp(pathname, arg0, arg1,...
        argn, NULL)
int execlpe(pathname, arg0, arg1,&ellip,
        argn, NULL, envp)
int execv(pathname, argv)
int execve(pathname, argv, envp)
int execvp(pathname, argv)
int execvpe(pathname, argv, envp)

  /* Path name of file to be run */
char *pathname;
  /* List of pointers to arguments */
char *arg0,*arg1,...,*argn;
  /* Array of pointers to arguments */
char *argv[ ];
  /* Array of pointers to environment settings */
char *envp[ ];
```

Comments:

The **exec** functions load and run new child processes. When the call is successful, the system places the child process in the storage previously occupied by the calling process. Sufficient storage must be available for loading and running the child process.

The *pathname* argument specifies the file to run as the child process. The *pathname* can specify a full path from the root, a partial path from the current working directory, or a file name. If *pathname* does not have a file name extension or does not end with a period (.), the **exec** functions first add the extension .COM and search for the file. (In OS/2 mode this search is not made.) If the search is unsuccessful, the **exec** functions try to find the file with the same file name and a .EXE extension. If *pathname* has an extension, the system uses only that extension. If *pathname* ends with a period, the **exec** functions search for *pathname* with no extension. The **execlp**, **execlpe**, and **execvp** functions search for the *pathname* in the directories that the PATH environment variable specifies.

You pass arguments 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. Do not include in the count the final null character (\0) for each string, but do count any space characters automatically inserted to separate arguments.

The compiler can pass the argument pointers as separate arguments (**execl**, **execle**, **execlp**, and **execlpe**) or as an array of pointers (**execv**, **execve** , **execvp**, and **execvpe**). You must pass at least one argument, *arg0* or *argv*[0], to the child process. By convention, this argument is a copy of the *pathname* argument, but a different value does not produce an error.

Use the **execl**, **execle**, **execlp**, and **execlpe** functions to call cases where you know the number of arguments in advance. The *arg0* argument is usually a pointer to *pathname*. The *arg1* through *argn* arguments are pointers to the character strings forming the new argument list. There must be a NULL pointer following *argn* to mark the end of the argument list. Use the **execv**, **execve**, **execvp**, and **execvpe** functions when the number of arguments to the new process is variable. Pass pointers to the arguments of these functions as an array, *argv*[ ]. The *argv*[0] argument is usually a pointer to *pathname*. The *argv*[1] through *argv*[*n*] arguments are pointers to the character strings forming the new argument list. A NULL pointer must follow *argv*[*n*] to mark the end of the argument list.

Files that are open when you make an **exec** call remain open in the new process. In the **execl**, **execlp**, **execv**, and **execvp** calls, the child process receives the environment of the parent. The **execle**, **execlpe**, **execvp**, and **execvpe** functions let you change the environment for the child process by passing a list of environment settings through the *envp* argument. The *envp* argument is an array of character pointers, each element of which points to a string, ending with a null character that defines an environment variable. Such a string usually has the following form:

NAME=value

where *NAME* is the name of an environment variable and *value* is the string value to which the **exec** function sets that variable.

Note: Do not enclose the *value* in double quotes.

The final element of the *envp* array should be NULL. When *envp* itself is NULL, the child process receives the environment settings of the parent process.

The **exec** functions do not normally return control to the calling process. They are equivalent to the corresponding **spawn** functions with P_OVERLAY as the value of *modeflag*. If an **exec** function returns, an error has occurred, and the return value is -1. The *errno* variable then has one of the following values.

Value	Meaning
E2BIG	The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K bytes.
EACCESS	The specified file has a locking or sharing violation.
EMFILE	There are too many open files. You must open the specified file to tell whether it is executable.
ENOENT	The file or *pathname* was not found, or was specified incorrectly in OS/2 mode.
ENOEXEC	The specified file cannot run or has an incorrect executable file format.
ENOMEM	One of the following conditions exists: • There is not enough storage available to run the child process. • The available storage has been corrupted. • An incorrect block exists, telling you that you did not properly reserve space for the parent process.

Example:

This example shows calls to four of the eight exec routines. When invoked without arguments from the command line, the program first runs the code for case PARENT. It then calls **exec()** to load and run a copy of itself. The instructions for the child are blocked to run only if *argv*[0] and one parameter were passed (case CHILD). In its turn, the child runs a child's child as a copy of the same program. The younger-generation child overlays the child in storage. This sequence is continued until four generations of child processes have run. Each of the processes prints a message identifying itself.

```
#include <stdio.h>
#include <process.h>
#define  PARENT       1
#define  CHILD        2

extern char **environ;
char *args[3];

main(argc, argv, envp)
int argc;
char **argv;
char **envp;
{
  switch(argc)
  {
     case PARENT:   /* No argument was
  passed:  run a child process.     */
     {
     printf("Parent process began.\n");
     execle(argv[0],argv[0],"1",NULL,envp);
     abort();  /*  Not executed because
             parent was overlaid.  */
     }
     case CHILD: /* One argument was passed:
          run a child's child.       */
     {
     printf("Child process %s began.\n",
             argv[1]);
        if('1'==*argv[1])
                   /* generation one */
        {
       execl(argv[0],argv[0],"2",NULL)
;
       abort();  /*  Not executed because
                 child was overlaid.  */
```

```
        }
        if('2'==*argv[1])
                  /* generation two */
        {
           args[0]=argv[0];
           args[1]="3";
           args[2]=NULL;
           execv(argv[0],args);
           abort();  /*  Not executed
        because child was overlaid.  */
        }
        if('3'==*argv[1])
                 /* generation three    */
        {
           args[0]=argv[0];
           args[1]="4";
           args[2]=NULL;
           execve(argv[0],args,environ);
           abort();    /*  Not executed
          because child was overlaid.*/
        }
        if('4'==*argv[1])
               /* generation four    */
            printf("Child process %s",
                    argv[1]);
    }
  }
  printf(" ended.\n");
  exit(0);
}
```

Related Topics:

abort, exit, _exit, spawnl, spawnle, spawnlp, spawnlpe, spawn, spawnve, spawnvp, spawnvpe, system

Note: The **exec** functions do not preserve the translation modes of open files. If the child process must use files received from the parent, use the **setmode** function to set the translation mode of these files to the desired mode. The **exec** functions do not preserve signal settings in child processes that calls to **exec** functions create. Calls to **exec** functions reset the signal settings to the default in the child process.

Do not use the **exec** functions if you plan to create a dual-mode executable using BIND. Use the corresponding SPAWN functions with *modeflag* set to P_WAIT.

exit - _exit

Purpose:

End the calling process.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>
#include <process.h>
  /* _exit defined in process.h */


  /* End after closing files */
void exit(status)
  /* End without flushing stream buffers */
void _exit(status)
int status;  /*Exit status */
```

Comments:

The **exit** and **_exit** functions end the calling process. They first call all functions that the **onexit** function has placed in a sequential list of functions, in reverse order. The **exit** function then flushes all buffers and closes all open files before ending the process. The **_exit** function ends the process without processing **onexit** functions or flushing stream buffers. The **exit** and **_exit** functions give the value 0 to *status* to show a normal exit and set *status* to some other value to show an error.

Although the **exit** and **_exit** calls do not return a value, they make the low-order byte of *status* available to the waiting parent process, if there is one, after the child process ends. If no parent process waits for the exiting process, the *status* value is lost. The *status* value is available to the DOS command IF ERRORLEVEL.

There is no return value.

Example:

The following example ends the process after flushing buffers and closing open files if it cannot open another file. Then, the example ends the process immediately if it cannot open a file.

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

FILE *stream;
main()
{
if ((stream = fopen("data","r")) == NULL) {
   perror("couldn't open data file");
   exit(1);
   }

if ((stream = fopen("data","r")) == NULL) {
   perror ("couldn't open data file");
   flushall();
   _exit(1);
   }
}
```

Related Topics:

abort, atexit, cwait, execl, execle, execlp, execv, execve, execvp, onexit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, system, wait

exp

Purpose:

Returns an exponential function of a floating-point argument.

Format:

#include <math.h>

double exp(*x*)
double *x*; /* Floating-point value */

Comments:

The **exp** function returns the exponential function of a floating-point argument *x*.

The **exp** function returns **e** to the power *x* (e^x). If an overflow occurs, the function returns HUGE_VAL; when an underflow occurs, it returns 0. Overflow also sets *errno* to ERANGE.

Example:

The following example calculates *y* as the exponential function of *x*:

```
#include <math.h>

double x, y;
       .
       .
       .
y = exp(x);
```

Related Topics:

log

Purpose:

Changes the size of a previously reserved storage block by expanding or contracting the block without moving its location in the heap.

Format:

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

void *_expand(ptr,size)
void *ptr;
size_t size;
```

Comments:

The **_expand** function changes the size of a previously reserved storage block by trying to expand or contract the block without moving its location in the heap. The *ptr* 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 *ptr* argument can also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **_expand**, **halloc**, **malloc**, or **realloc** since the block was freed. If *ptr* points to a freed block, the block remains free after a call to **_expand**.

The **_expand** function returns a pointer to the reallocated storage block. Unlike **realloc**, **_expand** cannot move a block to change its size. This means that the *ptr* argument to **_expand** is the same as the return value if there is sufficient storage available to expand the block without moving it.

The return value is NULL if there is not enough storage available to expand the block to the given size without moving it. In this case, the compiler has expanded, as much as possible in the current location, the item to which *ptr* points.

_expand

The storage space to which the return value points is aligned for storage of any type of object. You can check the new size of the item with the **_msize** function. To get a pointer to a type, use a type cast on the return value.

Example:

The following example reserves in storage a block of 10000 bytes and tries to expand it to a block of 64000 bytes, printing a report of the status of the operation.

```
#include <malloc.h>
#include <stdio.h>
main()
   {
   long *oldptr;
   size_t int newsize = 64000;

   oldptr = (long *)malloc(10000);
   printf("Size of storage block pointed to
           by oldptr = %u\n",
           _msize(oldptr));

   if (_expand(oldptr,newsize))
      printf("_expand was able to
              increase block to %u\n",
              _msize(oldptr));
   else
      printf("_expand was able to
              increase block to only %u\n",
              _msize(oldptr));
   }
```

Related Topics:

calloc, free, halloc, malloc, _msize, realloc

Purpose:

Returns the absolute value of a floating-point argument.

Format:

```
#include <math.h>

double fabs(x)
double x;   /* Floating-point value */
```

Comments:

The **fabs** function returns the absolute value of a floating-point argument. There is no error return value.

fabs

Example:

Compare the harmonic series expansion of the arctangent with the value returned by the library function. The number of terms to take the alternating series is given by the defined constant NTERM.

```
#define  NTERM     200
#include <math.h>
double atanterm();
main()
{
  double x;
  printf("Enter x, a floating point"
  " value, -1. < x < 1. : ");
  scanf("%E",&x);
  if(fabs(x)<=1.)
  {
    printf("\nTo %d terms, ",NTERM);
    printf("the harmonic series value"
    " of atan(%E)=%E\n",
    x,x*atanterm(x*x,1));
    printf("The C library function "
     "value of atan(%E)=%E\n",
     x,atan(x));
}
  else      printf("\nThe value of x "
 "must lie -1. < x < 1.\n");
 }

double atanterm(y,j)
double y;
int j;
{
   double z;

   z=y*j/(j+2);
   if(j > NTERM)
 return (1.);
   else
    return (1. - z * atanterm(y, j+2));
}
```

Related Topics:

abs, cabs, labs

Purpose:

Close open streams.

Format:
#include <stdio.h>

int fclose(*stream*) /* Close an open stream */
FILE **stream*; /* Pointer to file structure */

int fcloseall( ) /* Close all open streams */

Comments:

The **fclose** and **fcloseall** functions close a stream or streams. These functions flush all buffers associated with the streams before closing them. When they close the stream, these functions release the buffers that the system reserved. They do not automatically release buffers that the **setbuf** routine assigned.

The **fclose** function closes the given *stream*. The **fcloseall** function closes all open streams except the **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** streams, and any temporary files created by **tmpfile**.

The **fclose** function returns 0 if it successfully closes the stream. The **fcloseall** function returns the total number of streams closed. Both functions return EOF to indicate an error.

Example:

The following example opens a file *data* for reading as a data stream; then, it closes this file. It closes all other streams except the **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** data streams.

```
#include <stdio.h>

FILE *stream;
int numclosed;

stream = fopen("data", "r");
  .
  .
  .

  /* The following statement closes the stream. */
fclose(stream);

  /* The following statement closes all *
   * streams except stdin, stdout, *
   * stderr, stdaux, and stdprn. */
numclosed = fcloseall();
```

Related Topics:

close, fdopen, fflush, fopen, freopen

Purpose:

Converts a floating-point number to a character string.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *fcvt(value, ndec, decptr, signptr)
  /* Number to be converted */
double value;
  /* Number of digits after decimal point */
int ndec;
  /* Pointer to stored decimal point position */
int *decptr;
  /* Pointer to stored sign indicator */
int *signptr;
```

Comments:

The **fcvt** function converts a floating-point number to a character string. The *value* is the floating-point number to be converted. The **fcvt** function stores the digits of *value* as a string and adds a null character (\0). The *ndec* variable specifies the number of digits to be stored after the decimal point.

If the number of digits after the decimal point in *value* exceeds *ndec*, **fcvt** rounds the correct digit according to the FORTRAN F format. If there are fewer than *ndec* digits of precision, **fcvt** pads the string with zeros.

The **fcvt** function stores only digits in the string. You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr*. The *decptr* variable points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value shows that the decimal point lies to the left of the first digit. The *signptr* variable points to an integer showing the sign of *value*. The **fcvt** function sets the integer to 0 if *value* is positive and to a nonzero number if *value* is negative.

fcvt

The **fcvt** function returns a pointer to the string of digits. There is no error return value. Because of the limited precision of the double type, no more than 16 decimal digits are significant in any conversion.

Example:

This example will read in two floating-point numbers, compute their product, and print out only the billions digit of its character representation. At most, 16 decimal digits of significance can be expected.

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

main()
{
    float x, y;
    double z;
    int w, b, i, decimal, sign;
    char c;
    char *buffer;

    printf("Enter two floating-point "
    "numbers.\n");
    if(2!=(i=scanf("%e %e",&x,&y)))
    {
  printf("input error...\n");
  abort();
    }
    z=x*y;
    w=log10(fabs(z))+1.;
    buffer=fcvt(z,w,&decimal,&sign);
    b=decimal-10;
    if(b<0)
  printf("Their product does not"
      " exceed one billion.\n");
    if(b>15)
  printf("The billions digit of "
   "their product is insignificant.\n");
    if((b>-1)&&(b<16))
  printf("The billions digit of"
      " their product is %c.\n",
  buffer[b]);
}
```

Related Topics:

atof, atoi, atol, ecvt, gcvt

Note: The **ecvt** and **fcvt** functions use a single, statically-allocated buffer for the conversion. Each call to one of these functions destroys the result of the previous call.

fdopen

Purpose:

Associates an input or output stream with the file identified by a handle.

Format:

#include <stdio.h>

FILE *fdopen(*handle, type*)
int *handle*; /* Handle referring to open file */
char **type*; /* Type of access permitted */

Comments:

The **fdopen** function associates an input or output stream with the file identified by *handle*. The *type* variable is a character string specifying the type of access requested for the file.

Type	Description
"r"	Open a text file for reading. (The file must exist.)
"w"	Create a text file for writing. If the given file exists, erase its contents.
"a"	Open a text file for writing additional data at the end of the file. Create the file first if it does not exist.
"r+"	Open a text file for both reading and writing. (The file must exist.)
"w+"	Create a text file for both reading and writing. If the given file exists, erase its contents.
"a+"	Open a text file for reading and for writing additional data to the end of the file. Create the file first if it does not exist.
"rb"	Open a binary file for reading. The file must exist.
"wb"	Create a binary file for writing. If the given file exists, its contents are destroyed.

"ab" Open a binary file for writing at the end of that file. Create the file first if it does not exist.

"r+b" or "rb+"
Open a binary file for both reading and writing. The file must exist.

"w+b" or "wb+"
Create a binary file for both reading and writing. If the given file exists, **fdopen** erases its contents.

"a+b" or "ab+"
Open a binary file for both reading and adding to its end. Create the file first if it does not exist.

CAUTION:

Use the "w", "w+", "wb", "wb+", and "w+b" modes with care. They can destroy files.

The specified *type* must be compatible with the access mode you used to open the file.

When you open a file with "a" or "a+" as the value of *type*, all write operations take place at the end of the file. Although you can reposition the file pointer, using **fseek** or **rewind**, the file pointer always moves back to the end of the file before the system carries out any write operation. This action prevents you from writing over existing data.

When you specify any of the types containing "+" you can both read from and write to the file. The file is said to be open for "update." However, when switching from reading to writing, or the reverse, you must include an intervening **fseek** or **rewind** operation. You can specify the current position for the **fseek** operation.

In accessing text files carriage return-linefeed (CR-LF) combinations are translated into a single linefeed (LF) on input; linefeed characters are translated to CR-LF combinations on output. Accesses to binary files suppress all of these translations.

The **fdopen** function returns a pointer to the open stream. A NULL pointer value shows an error.

fdopen

Example:

The following example opens the file DATA and associates its file handle *fh* with the pointer *stream*.

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

FILE *stream;
int fh;

fh = open("data", O_RDONLY);

  /* The following statement associates a *
   * stream with the open file handle. */

stream = fdopen(fh, "r");
```

Related Topics:

dup, dup2, fclose, fcloseall, fopen, freopen, open

Purpose:

Tests the end-of-file indicator for a stream.

Format:

#include <stdio.h>

int feof(*stream*)
FILE **stream*; /* Pointer to a file structure */

Comments:

The **feof** function tells whether you have reached the end of the given *stream*. After you reach the end-of-file character, read operations return an end-of-file indicator until you close the stream or call **clearerr**, **fseek**, or **rewind**.

The **feof** function returns a nonzero value after the first read operation which attempts to read past the end-of-file character. The **feof** function returns the value 0 if no attempt has been made to read past the end-of-file character. There is no error-return value.

Example:

The following example scans the input file STREAM until it reads an end-of-file character.

```
#include <stdio.h>

char string[100];
FILE *stream;
void process(char *);
main()
  {

while (!feof(stream)) {
   if (fscanf(stream, "%s", string))
       process(string);}
   }
```

feof

Related Topics:

clearerr, eof, ferror, perror

Note: The **feof** routine is a macro. Since an empty file has no end-of-file character, calls to the **feof** for an empty stream always returns 0.

Purpose:

Tests for errors in reading from or writing to a stream.

Format:

#include <stdio.h>

int ferror(*stream*)
FILE **stream*; /* Pointer to file structure */

Comments:

The **ferror** function tests for an error in reading from or writing to the given *stream*. If an error occurs, the error indicator for the *stream* remains set until the you close *stream*, **rewind**, or call **clearerr**.

The **ferror** function returns a nonzero value to indicate an error on the given *stream*. A return value of 0 means no error has occurred.

Example:

The following example puts data out to a stream, and then checks to make sure a write error has not occurred. You must have previously opened the stream for writing.

```
#include <stdio.h>

FILE *stream;
char *string;
  .
  .
  .
fprintf(stream, "%s\n", string);
if (ferror(stream)) {
   printf("write error");
   clearerr(stream);
   }
```

ferror

Related Topics:

clearerr, eof, feof, fopen, perror

Note: The **ferror** routine is a macro.

Purpose:

Causes the system to write the contents of a buffer to a file.

Format:

#include <stdio.h>

int fflush(*stream*)
FILE **stream*; /* Pointer to a file structure */

Comments:

The **fflush** function causes the system to write the contents of the buffer associated with the specified output *stream* to a file. 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 **ungetch** for the *stream*. The *stream* remains open after the call. The **fflush** function has no effect on an unbuffered stream.

The **fflush** function returns the value 0 if it successfully flushes the buffer. It also returns the value 0 in cases where the specified stream has no buffer or is open for reading only. A return value of **EOF** shows an error.

Example:

The following statements flush a stream buffer and set up a new buffer for that stream.

```
#include <stdio.h>

FILE *stream;
char buffer[BUFSIZ];
  .
  .
  .
fflush(stream);
setbuf(stream,buffer);
```

fflush

Related Topics:

fclose, flushall, setbuf

Note: The system automatically flushes buffers when they are full, when you close the stream, or when a program ends normally without closing the stream.

Purpose:

Frees a storage block.

Format:

```
   /* Required only for function declarations */
#include <malloc.h>
void _ffree(ptr)
   /* Pointer to reserved storage block */
void far *ptr;
```

Comments:

The **_ffree** function frees a storage block outside the default data segment. *Ptr* points to a storage block previously reserved through a call to **_fmalloc**. The number of bytes freed is the number specified when the block was allocated. After the call, the freed block is again available. If *ptr* is NULL, **_ffree** ignores it.

Note: Attempting to free an incorrect *ptr* (a pointer not reserved with **_fmalloc**) can affect subsequent attempts to reserve storage and cause errors. In the small and medium models, a call to **free** is equivalent to a call to **_nfree**. In the compact and large models, a call to **free** is equivalent to a call to **_ffree**.

Example:

The following example illustrates the reserving and freeing of 100 bytes.

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

void far *alloc;

alloc = _fmalloc(100);
   .
   .
   .
if (alloc != NULL)  /* Test for a valid pointer */
   _ffree(alloc); /* Free storage for the heap */
```

_ffree

Related Topics:

_fmalloc, free, malloc

Purpose:

Read a single character from an input stream and increase the file pointer.

Format:

```
#include <stdio.h>
   /* Read a character from stream */
int fgetc(stream)
FILE *stream;  /* Pointer to file structure */

   /*  Read a character from  *
    * the standard input stream, stdin */
int fgetchar( );
```

Comments:

The **fgetc** function reads a single character from the input *stream* at the current position and increases the associated file pointer, if any, to point to the next character. The **fgetchar** function is the same as **fgetc(stdin)**.

The **fgetc** and **fgetchar** functions return the character read. A return value of EOF can show an error or end-of-file position; however, the EOF value is also a legitimate integer value. Use **feof** or **ferror** to tell whether the return value shows an error or an end-of-file position.

fgetc - fgetchar

Example:

The following example gathers a line of input from a stream. You can use **fgetchar( )** instead of **fgetc(stream)** in the **for** statement to gather a line of input from the **stdin** data stream. This operation is the same as **fgetc(stdin)**.

```
#include  <stdio.h>
.
.
.
    FILE *stream;
    char buffer[81];
    int  i, ch;
.
.
.
    for (i = 0; (i < 80) &&
     ((ch = fgetc(stream)) != EOF)
    && (ch != '\n'); i++)
    buffer[i] = ch;
    buffer[i] = '\0';
```

Related Topics:

fputc, fputchar, getc, getchar

Note: The **fgetc** and **fgetchar** functions are identical to **getc** and **getchar**, except that **fgetc** and **fgetchar** are functions, not macros.

Purpose:

Reads a string from the input data stream and stores it in a string.

Format:

```
#include <stdio.h>

  /* Read a string from stream */
char *fgets(string, n, stream)
char *string;  /* Storage location for data */
int n;  /* Number of characters stored */
FILE *stream; /* Pointer to file structure */
```

Comments:

The **fgets** function reads a string from the input *stream* and stores it in *string*. The **fgets** function reads characters from the current *stream* position up to and including the first new-line character (\n), up to the end of the stream, or until the number of characters read is equal to *n* -1, whichever comes first. The **fgets** function stores the result in *string* and adds a null character (\0) to the end of the string. The *string* includes the new-line character, if read. If *n* is equal to 1, the *string* is empty.

The **fgets** function returns the *string*. A NULL return value shows an error or an end-of-file condition. Use **feof** or **ferror** to determine whether the NULL value represents an error or the end of the file.

fgets

Example:

The following example gets a line of input from a data stream. The example reads no more than 99 characters, or up to \n, from the stream.

```
#include <stdio.h>

FILE *stream;
char line[100], *result;
  .
  .
  .
result = fgets(line,100,stream);
```

Related Topics:

fputs, gets, puts

Purpose:

Convert from IEEE single precision to binary or from binary to IEEE single precision.

Format:

```
#include <math.h>

   /* IEEE single to binary */
int fieeetomsbin(src4,dst4)
   /* Binary to IEEE single */
int fmsbintoieee(src4,dst4)

float *src4, *dst4;
```

Comments:

The **fieeetomsbin** routine converts a single-precision number in IEEE format to binary format. The **fmsbintoieee** routine converts a single-precision number in binary format to IEEE format.

The *src4* is a pointer to the **float** value to be converted. These functions store the result at the location given by *dst4*.

These functions return 0 if the conversion is successful and 1 if the conversion caused an overflow.

Example:

This example uses a particular value (-16101.) to test whether these two routines function as inverses of each other. After conversion and reconversion, the value -16101. is written out.

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

main()
{
   float b=-16101.;
   float  c, d;

   if (fmsbintoieee(&b, &c) == 1)
    fprintf(stderr, "overflow"
       " converting to IEEE form\n");
   if (fieeetomsbin(&c, &d) == 1)
    fprintf(stderr, "overflow"
       " converting from IEEE form\n");

   printf("The number after"
     " reconversion is %f\n",d);
}
```

Related Topics:

dieeetomsbin, dmsbintoieee

Note: These routines handle neither an IEEE NaN nor infinity. They treat IEEE denormals as 0 in the conversions. Numbers in the range 1.1755e-38 to 2.93874e-39, although representable as non-zero binary numbers, are IEEE denormals, which the compiler treats as 0 in the conversions.

Purpose:

Returns file length in bytes.

Format:

```
 /* Required for function declarations */
#include <io.h>

long filelength(handle)
int handle; /* Handle referring to an open file */
```

Comments:

The **filelength** function returns the length, in bytes, of the file associated with *handle*.

A return value of -1L indicates an error. The function will set *errno* to EBADF to show an incorrect file handle. If you have an open file to which you have appended bytes, you must close and reopen it before **filelength** can determine the updated length.

filelength

Example:

The following example tries to tell the length of the file associated with a data stream:

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

FILE *stream;
long length;

main()
{
      void flen();

      stream = fopen("flength.c","a");
      flen();
/*  Extend the file by five lines.     */
      fprintf(stream,"\n\n\n\n\n");
      flen();
      fclose(stream);

      stream = fopen("flength.c","r");
      flen();
}

void flen()
{
/*  Get length or -1L if function fails.   */
      length = filelength(fileno(stream));

      if (length == -1L)
   printf("filelength failed");
      else
   printf( "file length is %ld\n",length );
    return;
}
```

Related Topics:

chsize, fileno, fstat, stat

Purpose:

Returns the file handle currently associated with any data stream.

Format:

#include <stdio.h>

int fileno(*stream*)
FILE **stream*; /* Pointer to file structure */

Comments:

The **fileno** function returns the file handle currently associated with *stream*. If more than one handle is associated with the stream, the return value is the handle assigned when you first opened the stream.

There is no error return. The result is undefined if *stream* does not specify an open file.

Example:

The following example determines the file handle of the **stderr** data stream:

```
#include <stdio.h>

int result;

result = fileno(stderr); /* result is 2 */
```

Related Topics:

fdopen, filelength, fopen, freopen

Note: The **fileno** routine is a macro.

floor

Purpose:

Returns a floating-point value representing the largest integer less than or equal to *x*.

Format:

#include <math.h>

double floor(*x*)
double *x*; /* Floating-point value */

Comments:

The **floor** function returns the floating-point result as a double value. There is no error return.

Example:

This example computes *y* as the largest integer less than or equal to 2.8 and *z* as the largest integer less than or equal to -2.8.

```
#include <math.h>
double y, z;
  .
  .
  .
y = floor(2.8);    /* y = 2.0 */
z = floor(-2.8);   /* z = -3.0 */
```

Related Topics:

ceil, fmod

Purpose:

Causes the system to write the contents of buffers to the files.

Format:

#include <stdio.h>

int flushall( )

Comments:

The **flushall** function causes the system to write to files the contents of all buffers associated with open output streams. It clears all buffers associated with open input streams of their current contents. The next read operation, if there is one, then reads new data from the input files into the buffers. All streams remain open after the call.

The **flushall** function returns the number of open streams of input and output. There is no error-return value.

Example:

The following statement resolves any pending input or output on all streams:

```
#include <stdio.h>

int numflushed;
  .
  .
  .
numflushed = flushall();
```

Related Topics:

fflush

Note: The system automatically flushes buffers when they are full, when you close streams, or when a program ends normally without closing streams.

_fmalloc

Purpose:

Reserves a storage block outside the default data segment.

Format:

```
   /* Required only for function definitions */
#include <malloc.h>

void far *_fmalloc(size)
size_t size;  /* Bytes in reserved block */
```

Comments:

The **_fmalloc** function reserves a storage block of at least *size* bytes outside the default data segment. (The block might be larger than *size* bytes because of the space required for alignment and for maintenance information.)

The **_fmalloc** function returns a **far** pointer to the reserved space. The storage space to which the return value points is aligned for storage of any type of object. For a pointer to a type other than **void**, use a type cast on the return value.

If sufficient storage is not available outside the default data segment, **_fmalloc** tries to reserve a block of storage using the default data segment (near heap). If there is still not enough storage available, the return value is NULL.

Example:

The following example reserves space for 20 integers:

```
#include <malloc.h>

int far *intarray;

intarray = (int far *)_fmalloc(20*sizeof(int));
```

Related Topics:

_ffree, _fmsize, malloc, realloc

fmod

Purpose:

Calculates a floating-point remainder.

Format:

#include <math.h>

double fmod(*x*,*y*)
  /* Floating-point values */
double *x*;
double *y*;

Comments:

The **fmod** function calculates *f*, the floating-point remainder of *x* / *y*, such that $x = iy + f$ where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.

The **fmod** function returns the floating-point remainder. If *y* is zero or if *x* / *y* causes an overflow, the function returns 0.

Example:

The following example computes *z* as the remainder of *x*/*y*; here *x*/*y* is -3 with a remainder of -1:

```
#include <math.h>

double x, y, z;

x = -10.0;
y = 3.0;
z = fmod(x,y); /* z = -1.0 */
```

Related Topics:

ceil, fabs, floor

Purpose:

Returns the size, in bytes, of the storage block reserved by a call to **_fmalloc**.

Format:

#include <malloc.h>

size_t _fmsize(*ptr*)
void far **ptr*;

Comments:

The **_fmsize** function returns the size, in bytes, of the storage block reserved by a call to **_fmalloc**. The **_fmsize** function returns the size, in bytes, as an unsigned integer.

Example:

```
#include <malloc.h>

char far *stringarray;
unsigned int alloc_bytes;

stringarray = _fmalloc(200*sizeof(char));
if ((alloc_bytes = _fmsize(stringarray)) < 200)
   printf("Only %u bytes allocated\n",alloc_bytes);
else
   printf("All 200 bytes allocated\n");
```

Related Topics:

_ffree, **_fmalloc**, **malloc**, **_msize**, **_nfree**, **_nmalloc**, **_nmsize**

fopen

Purpose:

Opens a file.

Format:

#include <stdio.h>

FILE *fopen(*pathname, type*)
const char **pathname*; /* Path name of file */
const char **type*; /* Type of access permitted */

Comments:

The **fopen** function opens the file specified by *pathname*. The *type* variable is a character string specifying the type of access requested for the file, as follows:

Type	Description
"r"	Open a text file for reading. (The file must exist.)
"w"	Create a text file for writing. If the given file exists, its contents are destroyed.
"a"	Open a text file for writing at the end of that file. Create the file first if it does not exist.
"r+"	Open a text file for both reading and writing. (The file must exist.)
"w+"	Create a text file for both reading and writing. If the given file exists, **fopen** erases its contents.
"a+"	Open a text file for reading and appending.
"rb"	Open a binary file for reading. The file must exist.
"wb"	Create a binary file for writing. If the given file exists, its contents are destroyed.
"ab"	Open a a binary file for writing at the end of that file. Create the file first if it does not exist.

"r+b" or "rb+"
Open a binary file for both reading and writing. If the given file exists, **fopen** erases its contents.

"a+b" or "ab+"
Open a binary file for reading and appending. Create the file first if it does not exist.

"w+b" or "wb+"
Create a binary file for both reading and writing. If the given file exists, **fopen** erases its contents.

CAUTION:
Use the "w" "w+b", "wb+", "w+" and "wb" modes with care; they can destroy existing files.

When you open a file with "a" or "a+" type, all write operations take place at the end of the file. Although you can reposition the file pointer using **fseek** or **rewind**, these functions move the file pointer back to the end of the file before they carry out any write operation. Thus, you cannot write over existing data.

When you specify any of the types containing " + ", you can both read from and write to the file. The file is open for update. However, when switching between reading and writing, you must include an intervening **fseek** or **rewind** operation. You can specify the current position for the **fseek** operation.

In accesses to text files, carriage return/line feed combinations are translated into a single line feed on input; line feed characters are translated to carriage return/line feed combinations on output. Also, Ctrl+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, the function checks for a Ctrl+Z at the end of a file and removes it, if possible. Accesses to binary files suppress all of these translations.

The **fopen** function returns a pointer to the open file. A NULL pointer value indicates an error.

fopen

Example:

The following statements attempt to open a text file for reading.

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

FILE *stream;

if ((stream = fopen("data", "r")) == NULL)
   printf("couldn't open data file");
```

Related Topics:

fclose, fcloseall, fdopen, ferror, fileno, freopen, open, setmode

Note: You will not receive an error if you open the same file multiple times. However, if you write to a file using multiple handles, the file contents are unpredictable.

Purpose:

Return the offset and segment of the long pointer.

Format:

#include <dos.h>

unsigned FP_OFF(*longptr*)
unsigned FP_SEG(*longptr*)
  /* Long pointer to storage address */
char far **longptr*;

Comments:

The FP_OFF and FP_SEG macros return the offset and segment, respectively, of the long pointer *longptr.*

The FP_OFF macro returns an unsigned integer value representing an offset. The FP_SEG macro returns an unsigned integer value representing a segment address.

Under OS/2, references to segments are translated into selector values.

Example:

```
#include <dos.h>

char far *p;
unsigned int sp;
unsigned int op;
  .
  .
  .
sp = FP_SEG(p);
op = FP_OFF(p);
```

Related Topics:

segread

_fpreset

Purpose:

Resets the floating-point math package.

Format:

```
#include <float.h>

 /* Resets the floating-point math package */
void _fpreset( )
```

Comments:

The **_fpreset** function resets the floating-point math package. Usually, you use this function with **signal** or the **exec** or **spawn** family of routines.

If a program traps floating-point-error signals (SIGFPE) with **signal**, the program can safely recover from floating-point errors by calling **_fpreset** and doing a **longjmp**.

CAUTION:
Under DOS, a child process run by an exec, spawn, or system function might affect the floating-point state of the parent process if you use a numeric coprocessor. You should call _fpreset after any exec, spawn, or system call if the child process performed any operations using the numeric coprocessor.

Example:

The following example establishes the routine **fphandler** as a floating-point error handler. The main program produces a floating-point error, then calls **_fpreset** to reset the floating-point math package.

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

jmp_buf mark;


main()
{
    double a = 1.0, b = 0.0, c;
    int     fphandler();

    if (signal(SIGFPE, fphandler) ==
                          (int (*)()) -1)
        abort();
    if (setjmp(mark) == 0)
    {
        /* generate floating-point error */
        c = a / b;
        printf("Should never get here\n");
    }
    printf("Recovered from floating");
    printf("-point error\n");
}


int fphandler(sig, num)
int sig, num;
{
   printf("signal = %d, subcode = %d\n",sig, num);
/* Reinitialize floating-point package */
   _fpreset();
   longjmp(mark, -1);
}
```

Output:

```
signal = 8, subcode = 131
Recovered from floating-point error
```

Related Topics:

execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, signal, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe

fprintf

Purpose:

Formats and prints characters to the output stream.

Format:

```
#include <stdio.h>

int fprintf(stream, format-string[, argument...])
  /* Pointer to file structure */
FILE *stream;
  /* Format control string */
const char *format-string;
```

Comments:

The **fprintf** function formats and prints a series of characters and values to the output *stream*. The **fprintf** function converts each *argument*, if any, and puts out the stream according to the corresponding format specification in the *format-string*.

The *format-string* has the same form and function as the *format-string* argument for the **printf** function. See the **printf** reference page for a description of the *format-string* and the argument list.

The **fprintf** function returns the number of characters printed.

Example:

This example sends to the printer a line of asterisks for each integer in the array *count*[ ]. The number of asterisks printed on each line corresponds to an integer in the array.

```
#include <stdio.h>

int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};
main()
{
int i,j;
FILE *printer;

  /* Open the printer for writing */
printer = fopen("prn","w");
  /* Loop for each number in the array */
for (i=0; i < 10; i++)
{
     /* Loop for each count  */
   for (j = 0; j < count[i]; j++)
     /* Print asterisk */
   fprintf(printer,"*");
     /* Move to the next line */
   fprintf(printer,"\n");
}
fclose (printer);
}
```

Related Topics:

cprintf, fscanf, printf, sprintf

fputc - fputchar

Purpose:

Writes the character *c* to the output stream.

Format:

```
#include <stdio.h>

  /* Write a character to stream */
int fputc(c, stream)
int c;  /* Character to write */
FILE *stream;  /* Pointer to file structure */

int fputchar(c)  /* Write a character to STDOUT */
int c;  /* Character to write */
```

Comments:

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** functions return the character written. A return value of EOF can show an error. However, because the EOF value is also a legitimate integer value, use **ferror** to tell whether this is an error condition or the end of the file.

Note: The **fputc** and **fputchar** functions are equivalent to the **putc** and **putchar** macros.

Example:

The following example writes the contents of a buffer to a data stream.

Note: Because the output occurs as a side effect within the second expression of the **for** statement, the statement body is null. You can use **fputchar**(*buffer* [*i*]) instead of **fputc**(*stream*) in the **for** statement to write contents of the buffer to the **stdout** data stream. This statement has the same effect as **fputc**(*buffer*[*i*],**stdout**).

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;
  .
  .
  .
for (i = 0; (i < 81) &&
   ((ch = fputc(buffer[i],stream)) != EOF); i++);
```

Related Topics:

fgetc, fgetchar, putc, putchar

fputs

Purpose:

Copies a string to the output stream at the current position.

Format:

#include <stdio.h>

/* Write a string to *stream* */
int fputs(*string*, *stream*)
const char **string*; /* The string to put out */
FILE **stream*; /* Pointer to file structure */

Comments:

The **fputs** function copies *string* to the output *stream* at the current position. It does not copy the null character (\0) at the end of the string.

The **fputs** function returns the last character put out. If the input *string* is empty, the return value is 0. The return value EOF shows an error.

Example:

The following example writes a string to a stream.

```
#include <stdio.h>

FILE *stream;
int result;
  .
  .
  .
result = fputs("data files have been"
        "updated\n", stream);
```

Related Topics:

fgets, gets, puts

Purpose:

Reads items from the input stream and stores them in the buffer.

Format:

```
#include <stdio.h>

size_t fread(buffer, size, count, stream )
void *buffer; /* Storage location for data */
size_t size;   /* Item size in bytes */
size_t count; /* Maximum number of items to be read *
FILE * stream;  /* Pointer to file structure */
```

Comments:

The **fread** function reads up to *count* items of *size* length from the input *stream* and stores them in the given *buffer*. The file pointer associated with *stream*, if there is one, increases by the number of bytes read.

If the given *stream* was open in text mode, **fread** replaces carriage return/line feed characters with single line-feed characters. This replacement has no effect on the file pointer or the return value.

Under OS/2 in large and compact models, memory is reserved from the OS/2 heap. Each allocation unit is memory-protected and limited in size. In a **read** or **fread** call, if you give a read *count* that is greater than the size of the allocated buffer, OS/2 issues a **General Protection Failure** message, even if the file being read is small enough to fit within the boundaries of the buffer.

The **fread** function returns the number of full items actually read, which can be less than *count* if an error occurs or if the file end is met before reaching *count*.

Example:

The following statement reads 100 binary long integers from the stream.

```
#include <stdio.h>

FILE *stream;
long list[100];
int numread;

stream = fopen("data", "r+b");
  .
  .
  .
numread = fread((char *)list, sizeof(long),
  100, stream);
```

Related Topics:

fwrite, read

Purpose:

Frees a block of storage.

Format:

```
   /* Required for function declarations */
#include <stdlib.h>

void free(ptr)
   /* Pointer to a reserved block of storage */
void *ptr;
```

Comments:

The **free** function frees a block of storage. The *ptr* variable points to a block previously reserved with a call to **calloc**, **malloc**, or **realloc**. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of **realloc**) the block of storage. After the call, the freed block is available for reserving again. If *ptr* is NULL, **free** ignores it.

There is no return value.

Example:

The following example reserves 100 bytes and then frees them.

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

void *alloc;

alloc = malloc(100);
   .
   .
   .
if (alloc != NULL)   /* Test for a valid pointer */
    free(alloc);   /* Free storage for the heap */
```

Related Topics:

calloc, malloc, realloc

Note: Attempting to free an incorrect *ptr* (a pointer not allocated with **calloc**, **malloc**, or **realloc**) can affect the subsequent reserving of storage and cause errors. In the small and medium models, a call to **free** is equivalent to a call to **_nfree**. In the compact and large models, a call to **free** is equivalent to a call to **_ffree**.

Purpose:

Returns the number of items of a given size for which you can reserve storage in the default data segment.

Format:

#include <malloc.h>

unsigned int _freect(*size*)
size_t *size*; /* Item size in bytes */

Comments:

The **_freect** function tells you how much storage is available for dynamic allocation. The **_freect** function returns the number of items for which you can reserve storage in the default data segment.

The **_freect** function returns the number of items as an unsigned integer.

Example:

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

main()
   {
   printf("# of integers that can
   be dynamically allocated\n")
   printf("\t before malloc call = %u\n",
   _freect(sizeof(int)));
   malloc(500*sizeof(int));
   printf("# of integers that can
   be dynamically allocated\n");
   printf("\t after malloc call %u\n",
   _freect(sizeof(int)));
   }
```

_freect

Output:

(Actual numbers may vary slightly.)

```
# of integers that can be dynamically allocated
     before malloc call = 15312
# of integers that can be dynamically allocated
     after malloc call = 15059
```

Related Topics:

calloc, _expand, malloc, _memavl, _msize, realloc

Purpose:

Closes a file and reassigns a stream.

Format:

```
#include <stdio.h>

FILE* freopen(pathname, type, stream)
const char *pathname;   /* Path name of new file */
const char *type;   /* Type of access permitted */
FILE *stream;       /* Pointer to file structure *
```

Comments:

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *pathname*. You can use the **freopen** function to redirect the preopened files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** to files that you specify. The **freopen** function opens the new file associated with *stream* with the given *type*, which is a character string specifying the type of access requested for the file.

Type	Description
"r"	Open a text file for reading. (The file must exist.)
"w"	Create a text file for writing. If the given file exists, erase its contents.
"a"	Open a text file for writing additional data at the end of the file. Create the file first if it does not exist.
"r+"	Open a text file for both reading and writing. (The file must exist.)
"w+"	Create a text file for both reading and writing. If the given file exists, erase its contents.
"rb"	Open a binary file for reading. The file must exist.
"wb"	Create a binary file for writing. If the given file exists, its contents are destroyed.

"ab" Open a binary file for writing at the end of that file. Create the file first if it does not exist.

"r+b" or "rb+"
Open a binary file for both reading and writing. The file must exist.

"w+b" or "wb+"
Create a binary file for both reading and writing. If the given file exists, freopen erases its contents.

"a+b" or "ab+"
Open a binary file for both reading and adding to its end. Create the file first if it does not exist.

"a+" Open the file for reading and for writing additional data to the end of the file. Create the file first if it does not exist.

CAUTION:
Use the "w", "w+", "wb", "wb+ and "w+b" modes with care; they can destroy the contents of existing files.

When you open a file with "a" or "a+" as the value of *type*, all write operations take place at the end of the file. Although you can reposition the file pointer using **fseek** or **rewind**, these functions always move the file pointer back to the end of the file before carrying out any write operation. Thus, you cannot write over existing data.

When you specify any of the types containing "+", both reading and writing are allowed, and the file is open for updating. However, when switching between reading and writing, you must code an intervening **fseek** or **rewind** operation. You can specify the current position for the **fseek** operation.

In accesses to the text files, the **freopen** function translates carriage return/line feed characters to a single line-feed character when it puts in the data stream. It translates line feed characters to carriage return/line feed combinations when it puts out the data stream when accessing binary files.

The **freopen** function returns a pointer to the newly-opened file. If an error occurs, **freopen** closes the original file and returns a NULL pointer value.

Example:

After writing one line to the standard output file, this example closes stdout and opens a new file called altout. The output from the second printf call is directed to the altout file rather than to stdout.

```
#include <stdio.h>

FILE *stream1;
main()
{
 printf("This line goes to stdout.\n");
 stream1=freopen("altout","w",stdout);
 printf("This line goes to altout.\n");
}
```

Related Topics:

fclose, fcloseall, fdopen, fileno, fopen, open, setmode

frexp

Purpose:

Breaks down a floating-point value into a normalized fraction and an integer power of 2.

Format:

```
#include <math.h>

double frexp(x, expptr)
double x;    /* Floating-point value */
int *expptr; /* Pointer to stored integer exponent  */
```

Comments:

The **frexp** function breaks down the floating-point value *x* into a term *m* for the mantissa and another term *n* for the exponent, such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0 and $x = m*2$ to the power of *n*. The **frexp** function stores the integer exponent *n* at the location to which *expptr* points.

The **frexp** function returns the mantissa term *m*. If *x* is zero, the function returns 0 for both the mantissa and exponent. There is no error return value.

Example:

This example decomposes the floating-point value of *x*, 16.4, into its characteristic 0, its mantissa .5125, and its exponent 5. It stores the characteristic and mantissa in *y* and the exponent in *n*. The decomposition is computed in base 2.

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

main()
{
    int    n;
    double x, y;

    x = 16.4;
    /* y is .5125, n is 5   */
    y = frexp(x, &n);
    printf("x %lf, y %lf, n %d\n", x, y, n);
}
```

Related Topics:

ldexp, modf

fscanf

Purpose:

Reads data from a stream into specified locations.

Format:

#include <stdio.h>

int fscanf(*stream, format-string* [,*argument...*])
FILE **stream*; /* Pointer to file structure */
const char **format-string*; /* Format control string */

Comments:

The **fscanf** function reads data from the current position of the specified *stream* into the locations given by the *arguments*, if any. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function. See the **scanf** reference page for a description of the *format-string*.

The **fscanf** function returns the number of fields that it successfully converted and assigned. The return value does not include fields that **fscanf** read but did not assign.

The return value is EOF for an attempt to read at end-of-file. A return value of 0 means that **fscanf** assigned no fields.

Example:

This example opens the file DATA for reading and then scans this file for a string, a character, a long integer value, and a floating-point value.

```
#include <stdio.h>

FILE *stream;
long l;
float fp;
char s[81];
char c;

stream = fopen("data", "r");
  .
  .
  .
   /* Put in various data. */

fscanf(stream, "%s", s);
fscanf(stream, "%c", &c);
fscanf(stream, "%ld", &l);
fscanf(stream, "%f", &fp);
```

Related Topics:

cscanf, fprintf, scanf, sscanf

fseek

Purpose:

Moves the file pointer to a new location.

Format:

#include <stdio.h>

int fseek(*stream*, *offset*, *origin*)
FILE **stream*; /* Pointer to file structure *
long int *offset*; /* Number of bytes from *origin* */
int *origin*; /* Initial position */

Comments:

The **fseek** function moves any file pointer, associated with *stream* to a new location that is *offset* bytes from the *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 reading or a writing operation.

The *origin* must be one of the following constants, defined in **stdio.h**:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file.

The **fseek** function can reposition the pointer anywhere in a file. You can also position the pointer beyond the end of the file. However, an attempt to position the pointer before the beginning of the file causes an error. The **fseek** function clears the end-of-file indicator, even when *origin* is SEEK_END.

The **fseek** function returns the value 0 if it successfully moves the pointer. A nonzero return value shows an error. On devices that cannot seek, such as screens and printers, the return value is undefined.

Example:

The following example opens a file DATA for reading. After performing input operations (not shown), it moves the file pointer to the beginning of the file.

```
#include <stdio.h>
main()
{
FILE *stream;
int result;

stream = fopen("data", "r");
  .
  .
  .
result = fseek(stream, 0L, SEEK_SET);
}
```

Related Topics:

ftell, lseek, rewind

Note: For *streams* opened in text mode, **fseek** has limited use because carriage return/line feed translations can produce unexpected results. The only **fseek** operations that work on *streams* opened in text mode are seeking with an offset of zero relative to any of the origin values or seeking from the beginning of the file with an offset value returned from a call to **ftell**.

fstat

Purpose:

Obtains information about an open file.

Format:

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

int fstat(handle, buffer)
  /* Handle referring to open file */
int handle;
  /* Pointer to structure to store results */
struct stat *buffer;
```

Comments:

The **fstat** function obtains information about the open file associated with the given *handle* and stores it in the structure to which *buffer* points. The **sys\stat.h** include file defines the **stat** structure. The **stat** structure contains the following fields:

Field	Value
st_mode	Bit mask for file mode information. The **fstat** function sets the S_IFCHR bit if *handle* refers to a device. It sets the S_IFREG bit if *handle* refers to an ordinary file. It sets user read/write bits according to the permission mode of the file.
st_dev	The drive number of the disk containing the file or *handle* for a device. This field is not defined under OS/2.
st_rdev	The drive number of the disk containing the file or *handle* for a device. (This is the same as **st_dev**. It is not defined under OS/2.)
st_nlink	Always 1.
st_size	The size of the file in bytes.

st_atime The time of the last change in the file. (This is the same as **st_mtime** and **st_ctime**.)

st_mtime The time of the last change in the file. (This is the same as **st_atime** and **st_ctime**.)

st_ctime The time of last change in file. (This is the same as **st_atime** and **st_mtime**.

There are three additional fields in the **stat** structure type that do not contain meaningful values under DOS.

The **fstat** function returns the value 0 if it obtains the file status information. A return value of -1 indicates an error; in this case, **fstat** sets *errno* to EBADF, showing an incorrect file handle.

Example:

This program uses **fstat** to report the size of a file named DATA.

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

struct stat buf;
int fh, result;
char *buffer = "A line to output";
main()
{
fh = open("tmp\\data", O_CREAT |
    O_WRONLY |O_TRUNC);
write (fh, buffer, strlen(buffer));
  .
  .
  .
result = fstat(fh, &buf);

if (result == 0) {
   printf("file size is %ld\n",
 buf.st_size);
   printf("drive number is %d\n",
 buf.st_dev);
   printf("time modified is %s\n",
   ctime(&buf.st_atime));
   }
else printf("Bad file handle\n");
}
```

fstat

Related Topics:

access, chmod, filelength, stat

Note: If the given *handle* refers to a device, the size and time fields in the **stat** structure are not meaningful.

Purpose:

Gets the current position of the file pointer.

Format:

```
#include <stdio.h>

   /* Pointer to file structure */
long int ftell(stream)
FILE *stream;
```

Comments:

The **ftell** function gets the current position of any file pointer associated with *stream*. The **ftell** function expresses the position as an offset relative to the beginning of the *stream*.

The **ftell** function returns the current position. On error, **ftell** returns -1L and sets errno to a non-zero value.

Possible values for errno are:

EBADF Incorrect file number.

EINVAL Non-valid stream argument.

On devices that cannot seek, such as screens and printers, or when *stream* does not refer to an open file, the return value is undefined.

ftell

Example:

The following example opens the file DATA for reading. After performing input operations (not shown), it puts the current file pointer position into the **long** variable *position.*

```
#include <stdio.h>

FILE *stream;
long position;

stream = fopen("data", "rb");
  .
  .
  .
position = ftell(stream);
```

Related Topics:

fseek, lseek, tell

Note: Text mode causes translation of carriage return and line feed characters. The value returned by **ftell** might not reflect the offset, in bytes, of the physical position of the file pointer in streams opened in text mode. Use **ftell** in conjunction with the **fseek** function to remember and return to file locations correctly.

The current file position returned by **ftell** is not necessarily the position at which the next write operation would occur. For example, when a file is opened for appending, **ftell** returns a position at the beginning of the file, even though the next write operation would occur at the end.

Purpose:

Gets the current time and stores it.

Format:

```
#include <sys\types.h>
#include <sys\timeb.h>

void ftime(timeptr)
 /* Pointer to structure defined in sys\timeb.h */
struct timeb *timeptr;
```

Comments:

The **ftime** function gets the current time and stores it in the structure to which *timeptr* points. The **sys\timeb.h** include file contains the definition of the **timeb** structure. It contains four fields: *time*, *millitm*, *timezone*, and *dstflag*.

The *time* field gives the time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. The *millitm* field gives the fraction of a second, in milliseconds. The *timezone* field gives the difference in minutes between Greenwich Mean Time and local time, moving westward. The **ftime** function sets the value of *timezone* from the value of the global variable *timezone* (see **tzset**). The *dstflag* is nonzero if Daylight Saving Time is currently in effect for the local time zone. For an explanation of how daylight savings time is determined, see **tzset** in this chapter.

The **ftime** function gives values to the fields in the structure to which *timeptr* points. It does not return a value.

ftime

Examples:

This example polls the system clock, converts the current time to a character string, and prints the string. The following is the format of the output:

```
the time is Tues May 05 10:13:55 1987
```

This example saves the time data in the structure *timebuffer*.

```
#include <sys\types.h>
#include <sys\timeb.h>
#include <stdio.h>
#include <time.h>

struct timeb timebuffer;

ftime(&timebuffer);
printf("the time is %s\n",
       ctime(&(timebuffer.time)));
```

Related Topics:

asctime, ctime, gmtime, localtime, time, tzset

Purpose:

Writes items to the output stream.

Format:

```
#include <stdio.h>

size_t fwrite(buffer, size, count, stream)
  /* Pointer to data to be written */
const void *buffer;
size_t size;   /* Item size in bytes */
  /* Maximum number of items to be written */
size_t count;
FILE *stream;        /* Pointer to file structure *
```

Comments:

The **fwrite** function writes up to *count* items of *size* length from *buffer* to the output *stream*. The file pointer associated with *stream*, if there is a pointer, increases by the number of bytes actually written.

If the given *stream* is open in text mode, **fwrite** replaces each carriage return with a carriage-return/line feed pair. The replacement has no effect on the return value.

The **fwrite** function returns the number of full items actually written, which can be less than *count* if an error occurs.

fwrite

Example:

The following example writes 100 long integers to a stream in binary format.

```
#include <stdio.h>

FILE *stream;
long list[100];
int numwritten;

stream = fopen("data", "w+b");
  .
  .
  .
numwritten = fwrite((char *)list, sizeof(long),
      100, stream);
```

Related Topics:

fread, write

Purpose:

Converts a floating-point value to a character string and stores the string in a buffer.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *gcvt(value, ndec, buffer)
double value;   /* Value to convert */
int ndec;  /* Number of significant digits stored */
char *buffer;   /* Storage location for result */
```

Comments:

The **gcvt** function converts a floating-point *value* to a character string and stores the string in the *buffer*. The *buffer* should be large enough to hold the converted value and a null character (\0), which **gcvt** automatically adds to the end of the string. There is no provision for overflow.

The **gcvt** function tries to produce *ndec* significant digits in FORTRAN F format. Failing that, it produces *ndec* significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion.

A FORTRAN F number has the following format:

[*sign*]*digit*[*digit* . . .] .[*digit*]

A FORTRAN E number has the following format:

[*sign*]*digit.digit*[*digit* . . .] **E**[*sign*]*digit digit*

The **gcvt** function returns a pointer to the string of digits. There is no error return value.

gcvt

Example:

The following example converts the value -3.1415e5 to a character string and places it in the character array buffer.

```
#include <stdlib.h>

char buffer[50];
int precision = 7;
  /* Buffer contains "-314150.0\0" */
gcvt(-3.1415e5, precision, buffer);
```

Related Topics:

atof, atoi, atol, ecvt, fcvt

Purpose:

Reads a single character and increases the file pointer.

Format:

```
#include <stdio.h>

  /* Read a character from stream */
int getc(stream)
FILE *stream;  /* Pointer to file structure */

int getchar( );  /* Read a character from stdin */
```

Comments:

The **getc** function reads a single character from the current *stream* position and increases the associated file pointer, if there is one, to point to the next character. The **getchar** function is identical to **getc(stdin)**.

The **getc** and **getchar** functions return the character read. A return value of EOF shows an error or end-of-file condition. Use **ferror** or **feof** to determine whether an error or an end-of-file condition occurred.

getc - getchar

Example:

The following example gets a line of input from the **stdin** data stream. You can also use **getc(stdin)** instead of **getchar()** in the **for** statement to get a line of input from **stdin**.

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i, ch;
  .
  .
  .
for (i = 0; (i < 80) && ((ch = getchar()) != EOF)
     &&(ch != '\n'); i++)
   buffer[i] = ch;

buffer[i] = '\0';
```

Related Topics:

fgetc, fgetchar, getch, getche, putc, putchar, ungetc

Note: The **getc** and **getchar** routines are identical to **fgetc** and **fgetchar**, but are macros, not functions.

Purpose:

Reads a single character from the keyboard without echoing it on a screen or printer.

Format:

```
  /* Required for function declarations */
#include <conio.h>

int getch( );
```

Comments:

The **getch** function reads, without echoing, a single character directly from the keyboard. If you type Ctrl-C, the system ends your program.

The **getch** function returns the character read. In case of error in the OS/2 mode, **getch** returns EOF.

Example:

The following example gets characters from the keyboard until it finds a non-blank character.

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

int ch;
 /* This loop gets characters until a nonblank *
  * character is seen.  Preceding blanks are   *
  * discarded.                                 */
do {
   ch = getch();
   }
   while (isspace(ch));
```

Related Topics:

cgets, getche, getchar

getche

Purpose:

Reads a single character from the keyboard and echoes it on a screen or printer.

Format:

```
  /* Required for function declarations */
#include <conio.h>

int getche( )
```

Comments:

The **getche** function reads a single character from the keyboard and displays the character read. Under DOS, if you type Ctrl+Break, the system performs a DOS INT 23H to end your rogram.

The **getche** function returns the character read. There is no error return value.

Example:

The following example gets a character from the keyboard and echoes it to the screen. If the character is an uppercase letter, **getche** converts it to lowercase and writes over the old character.

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

int ch;

ch = getche();
if (isupper(ch))
   cprintf("\b%c", _tolower(ch));
```

Related Topics:

cgets, getch, getchar

Purpose:

Gets the full pathname of the current working directory.

Format:

```
  /* Required for function declarations */
#include <direct.h>

char *getcwd(pathbuf, n)
  /* Storage location for pathname */
char *pathbuf;
int n;  /* Maximum length of pathname */
```

Comments:

The **getcwd** function gets the full pathname of the current working directory and stores it at *pathbuf*. The integer argument *n* specifies the maximum length for the pathname. An error occurs if the length of the pathname (including the terminating null character) exceeds *n*.

The *pathbuf* argument can be NULL; **getcwd** will reserve a buffer of at least *n* bytes (using **malloc**) to store the pathname. If the current working directory string is more than *n* bytes, the reserved buffer will be large enough to contain the string. You can later free this buffer using the **getcwd** return value as a pointer to the buffer in the **free** function.

The **getcwd** function returns *pathbuf*. A NULL return value indicates an error and sets *errno* to one of the following values.

Value	Meaning
ENOMEM	Not enough storage space available to reserve *n* bytes (when NULL argument is *pathbuf*)
ERANGE	Path name longer than *n* characters.

getcwd

Example:

The following example stores the name of the current working directory (up to 128 characters) in a buffer.

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

char buffer[129];

if (getcwd(buffer, 128) == NULL)
   perror("getcwd error");
```

Related Topics:

chdir, mkdir, rmdir

Purpose:

Searches environment variables for *varname*.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *getenv(varname);
const char *varname;  /* Name of environment variable */
```

Comments:

The **getenv** function searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment (for example, the default search path for libraries linked with a program) in which a process runs.

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.

Example:

The following example gets the value of the PATH environment variable. If an entry such as PATH=A:\BIN;B:\SUE is in the environment, *pathvar* points to A:\BIN;B:\SUE.

```
#include <stdlib.h>

char *pathvar;

pathvar = getenv("PATH");
```

getenv

Related Topics:

putenv

Note: Do not directly change environment table entries. To change an entry, use the **putenv** function. To change 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 get access to the environment table. The **putenv** function can change the value of *environ*, thus invalidating the *envp* argument to the **main** function.

Purpose:

Returns the process identification.

Format:

```
   /* Required for function declarations */
#include <process.h>

int getpid( )
```

Comments:

The **getpid** function returns an integer, the process identification, which uniquely identifies the calling process. There is no error return value.

Example:

The following example prints **FILE***xxxxx*, where *xxxxx* is the process identification.

```
#include <process.h>
#include <string.h>
#include <stdio.h>
char filename[11], pid[6];
         .
         .
         .
strcpy(filename, "FILE");
strcat(filename, itoa(getpid(),pid,10));
printf("Filename is %s\n", filename);
```

Related Topics:

mktemp

gets

Purpose:

Reads a line from the standard input-data stream **stdin**.

Format:

#include <stdio.h>

char *gets(*buffer*)
  /* Storage location for input string */
char **buffer*;

Comments:

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.

The **gets** function returns its argument. A NULL pointer shows an error or an end-of-file condition. Use **ferror** or **feof** to tell which of these conditions occurred.

Example:

The following statement gets a line of input from **stdin**:

```
#include <stdio.h>

char line[100];
char *result;

result = gets(line);
```

Related Topics:

fgets, fputs, puts

Purpose:

Reads the next binary value from a stream and increases the file pointer.

Format:

#include <stdio.h>

int getw(*stream*)
FILE **stream*; /* Pointer to file structure */

Comments:

The **getw** function reads the next binary value of type **int** from the specified input *stream* and increases the associated file pointer, if there is one, to point to the next unread character. The **getw** function does not assume any alignment of items in the *stream*.

The **getw** function returns the integer value read. A return value of EOF can show an error or the end of the file, but the EOF value is also a legitimate integer value. Use **feof** or **ferror** to verify an whether this return value is an end-of-file or error condition.

Example:

The following example reads a binary word from the input *stream*. If it finds an error, it prints **getw failed** and resets the error flag for the stream.

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

FILE *stream;
int i;
  .
  .
  .
i = getw(stream);

if (ferror(stream)) {
   printf("getw failed");
   clearerr(stream);
   }
```

getw

Related Topics:

putw

Note: IBM provides the **getw** function primarily for compatibility with previous libraries. Portability problems can occur with **getw** because the size of an **int** and ordering of bytes within an **int** differ between systems.

Purpose:

Converts *time*, a **long** integer variable, to a structure variable.

Format:

```
#include <time.h>

struct tm *gmtime(time)
const time_t *time;   /* Pointer to stored time */
```

Comments:
The **gmtime** function converts a time value to a structure. The value *time* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich Mean Time; this value is usually obtained from a call to **time**.

The **gmtime** function breaks down the *time* value and stores it in a **tm** structure, defined in **time.h**. The structure reflects Greenwich Mean Time, not local time.

The fields of the **tm** structure store those values:

Field	Value Stored
tm_sec	Seconds
tm_min	Minutes
tm_hour	Hours (0-24)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Always 0 for **gmtime**. For **localtime** this value is nonzero if daylight savings time is in effect, and zero otherwise.

gmtime

DOS does not understand dates prior to 1980. If *time* represents a date before January 1, 1980, **gmtime** returns NULL.

The **gmtime** function returns a pointer to the structure result. There is no error return value.

Example:

This program uses the **gmtime** function 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 <stdio.h>
#include <time.h>

struct tm *newtime;
time_t ltime;
main()
{
time(&ltime);
newtime = gmtime(&ltime);
printf ("Greenwich Mean Time is %s\n",
        asctime(newtime));
}
```

Related Topics:

asctime, ctime, ftime, localtime, time, tzset

Note: The **gmtime** and **localtime** functions use a single, statically-allocated structure to hold the result. Each call to one of these functions destroys the result of the previous call.

Purpose:

Reserves storage for huge arrays.

Format:

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

void huge *halloc(n,size)
long n;  /* Number of elements */
   /* Length in bytes of each element */
size_t size;
```

Comments:

The **halloc** function call DOS to reserve storage space for a huge array of *n* elements, each of length *size* bytes. **Halloc** resets each element to 0.

If the size of the array is greater than 128K bytes, the size of an array element must be a power of 2.

Halloc returns a **huge** pointer to the reserved space. The storage space to which the return value points is aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value. The return value is NULL if there is not enough storage space available, or if the huge array has been specified illegally.

halloc

Example:

This example allocates space for 80000L long integers and sets the space to zero.

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

main()
{
    long huge *lalloc;
    lalloc =
       (long huge *)halloc(80000L,sizeof(long));
    if (lalloc == NULL)
        printf("Insufficient memory available");
    else
        printf("Memory successfully allocated");
}
```

Related Topics:

calloc, free, hfree, malloc, realloc, _fmalloc,_nmalloc

Note: Unlike the related memory allocation functions **_fmalloc** and **_nmalloc**, **halloc** performs no heap management. It allocates strictly on the basis of what storage is available from DOS.

Purpose:

Frees a block of space in storage.

Format:

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

void hfree(ptr)
   /* Pointer to reserved storage block */
void huge *ptr;
```

Comments:

The **hfree** function frees a block of space in storage. The *ptr* points to a storage block previously reserved through a call to **halloc**. The number of bytes freed is the number of bytes specified when you reserved the block. After the call, the freed block is available storage.

Note: Attempting to free an incorrect *ptr* (a pointer not reserved with **halloc**) can affect subsequent allocation and can cause errors. The **hfree** function performs no heap management. It merely gives back to DOS the allocated block.

hfree

Example:

The following example reserves 80000 bytes and then frees them.

```
#include <malloc.h>

void huge *alloc;

alloc = halloc(80000L,sizeof (char));
   .
   .
   .
if (alloc != NULL)      /* Test for valid pointer */
   hfree(alloc); /* Free memory for the heap */
```

Related Topics:

halloc

Purpose:

Calculates the length of the hypotenuse.

Format:

#include <math.h>

double hypot(*x*,*y*)
double *x*, *y*; /* Floating-point values */

Comments:

The **hypot** function calculates the length of the hypotenuse of a right triangle based on the lengths of two sides *x* and *y*. A call to **hypot** is equal to:

sqrt(*x*x* + *y*y***);**

The **hypot** function returns the length of the hypotenuse. If an overflow results, the function sets *errno* to ERANGE and returns the value HUGE_VAL.

Example:

The following example calculates the hypotenuse of a right triangle with sides of 3.0 and 4.0.

```
#include <math.h>

double x, y, z;

x = 3.0;
y = 4.0;
z = hypot(x,y); /* z = 5.0 */
```

Related Topics:

cabs

inp

Purpose:

Reads one byte from the input port.

Format:

```
  /* Required for function declarations */
#include <conio.h>

int inp(port)
unsigned port; /* Port number */
```

Comments:

The **inp** function reads one byte from the input port specified by *port*. The *port* argument can be any unsigned integer number in the range 0 to 65535. There is no error return value.

Example:

The following example reads a byte from the port to which *port* is currently set.

```
#include <conio.h>
main()
{
unsigned port=0x64;
char result;
result = inp(port);
printf("%0x\n",(int)result);
}
```

Related Topics:

outp

Note: Under OS/2, **inp** uses privileged **IN** instructions that require you to set up your system to get access to the input/output privilege level (IOPL) and to provide a .DEF file for your program. Use **inp** only for getting access to ports for graphics adapters. For more information on IOPL, see the *IBM Operating System/2 Technical Reference* book.

Purpose:

Performs an 8086 software interrupt.

Format:

```
#include <dos.h>

int int86(intno, inregs, outregs)
int intno;    /* Interrupt number */
   /* Register values on call */
union REGS *inregs;
   /* Register values on return */
union REGS *outregs;
```

Comments:

Under DOS, the **int86** function performs the 8086 software interrupt specified by the interrupt number *intno*. Before performing the interrupt, **int86** copies the contents of *inregs* to the corresponding registers. After the interrupt returns, the function copies the current register values to *outregs*. It also copies the status of the system carry flag to the *cflag* field in *outregs*. The *inregs* and *outregs* arguments are unions of type REGS. The include file **dos.h.** defines the union type REGS.

Use the **int86** function to perform DOS interrupts directly. The **int86** function is not available under OS/2. If you use **int86** for OS/2, you receive an unresolved external reference at link time.

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, and the *doserrno* variable is also set to the corresponding error code.

Example:

This program uses the **int86** to call the bios video service (INT [10]) to change the size of the cursor.

The default values are:

Monochrome card: 12 13
Color card: 6 7
43-line EGA: 4 5

```
#define VIDEO_IO  0x10
#define SET_CRSR  1

#include <dos.h>
#incluse <stdio.h>

union REGS regs;

main ()
{
 int top, bot;

 printf ("Enter new cursor top and bottom: ");
                              /* Get new cursor size from user */
 scanf ("%d %d", &top, &bot);

regs.h.ah = SET_CRSR; /* Set up for cursor
 change call */
regs.h.ch = top;
regs.h.cl = bot;

int86 (VIDEO_IO, &regs, &regs);  /*Execute interrupt  */
}
```

Related Topics:

bdos, intdos, intdosx, int86x

Purpose:

Performs an 8086 software interrupt.

Format:

```
#include <dos.h>

int int86x(intno,inregs,outregs,segregs)
int intno;
  /* Register values on call */
union REGS *inregs;
  /* Register values on return */
union REGS *outregs;
  /* Segment register value on call */
struct SREGS *segregs;
```

Comments:

Under DOS, the **int86x** function performs the 8086 software interrupt specified by the interrupt number *intno*. Unlike the **int86** function, **int86x** accepts segment register values in *segregs*, letting programs that use large model data segments or far pointers specify the segment or pointer used during the system call. Before performing the specified interrupt, **int86x** copies the contents of *inregs* and *segregs* to the corresponding registers. The **int86x** function uses only DS and ES register values in *segregs*. After the interrupt returns, the function copies the current register values to *outregs*, copies the current ES and DS values to *segregs*, and restores DS. It also copies the status of the system carry flag to the *cflag* field in *outregs*. The *inregs* and *outregs* arguments are unions of type REGS. The *segregs* argument is a structure of type SREGS. The include file **dos.h** defines these types.

Use the **int86x** function to call DOS interrupts that take an argument in the ES register or take a DS register value that is different from the default data segment.

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,

and **int86x** sets the *doserrno* variable to the corresponding error code.

This function is not available under OS/2.

Example:

The following example uses the **int86x** function to produce software interrupt 0x21, which then makes the DOS CHANGE ATTRIBUTES system call. This example uses the **int86x** function because the filename to which it refers can be in a segment other than the default data segment. (A far pointer is the reference to it.) You must explicitly set the DS register with the SREGS structure.

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

  /* INT 21H makes system calls */
#define SYSCALL  0x21
  /* System call 43h - Change Attributes */
#define CHANGE_ATTR 0x43

  /* The filename is in a 'far' data segment */
char far *filename = "int86x.c";

union REGS inregs, outregs;
struct SREGS segregs;
int result;
main()
{
   /* AH is system call number */
inregs.h.ah = CHANGE_ATTR;
   /* AL is function (get attributes) */
inregs.h.al = 0;
   /* DS:DX points to filename */
inregs.x.dx = FP_OFF(filename);
segregs.ds = FP_SEG(filename);
result = int86x(SYSCALL, &inregs, &outregs,
  &segregs);
if (outregs.x.cflag) {
   printf("can't get file attributes;
   error number %d\n",result);
   exit(1);
      }
else
  printf("Attribs = %#x\n", outregs.x.cx);
}
```

Output:

If the file is open only for reading, the following output is displayed:

```
Attribs = 0x0001
```

Related Topics:

bdos, intdos, intdosx, int86, segread, FP_SEG

Note: You can obtain segment values for the *segregs* argument by using either the **segread** function or the FP_SEG macro.

intdos

Purpose:

Makes a DOS system call.

Format:

```
#include <dos.h>

int intdos(inregs, outregs)
  /* Register values on call */
union REGS *inregs;
  /* Register values on return */
union REGS *outregs;
```

Comments:

The **intdos** function makes 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 **dos.h** include file defines the REGS union type.

To make a system call, **intdos** performs a DOS INT 21H instruction. Before performing the instruction, the function copies the contents of *inregs* to the corresponding registers. After the DOS INT instruction returns, **intdos** copies the current register values to *outregs*. It also copies the status of the system carry flag to the *cflag* field in *outregs*. If this field is nonzero, the system call sets the flag to show an error condition.

Use the **intdos** routine to make DOS system calls that take arguments in registers other than DX (DH/DL) and AL or to make system calls that show errors by setting the carry flag.

The **intdos** function returns the value of the AX register after the system call is complete. If the *cflag* field in *outregs* is is nonzero, an error has occurred. The **intdos** function sets *_doserrno* to the corresponding error code.

This function is not available under OS/2. For information on calling OS/2 functions from a C program, see "Application Program Interface" in the *IBM Operating System/2 Technical Reference* book.

Example:

Suppose that the current date is March 31, 1986. In the following example, the DOS INT 2AH system call makes the date available for printing as "**date is 3/31/1986**".

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

union REGS inregs, outregs;
main()
{
     .
     .
inregs.h.ah = 0x2a;
intdos(&inregs, &outregs);
printf("date is %d/%d/%d\n", outregs.h.dh,
   outregs.h.dl, outregs.x.cx);
}
```

Related Topics:

bdos, intdosx

intdosx

Purpose:

Makes a DOS system call.

Format:

```
#include <dos.h>

int intdosx(inregs, outregs, segregs)
  /* Register values on call */
union REGS *inregs;
  /* Register values on return */
union REGS *outregs;
  /* Segment register values on call */
struct SREGS *segregs;
```

Comments:

The **intdosx** function makes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. Unlike the **intdos** function, **intdosx** accepts segment register values in *segregs*, letting programs that use long model data segments or far pointers specify which segment or pointer should be used during the system call. The *inregs* and *outregs* arguments are unions of type REGS. The *segregs* argument is a structure of type SREGS. The include file **dos.h** defines these types.

To make a system call, **intdosx** performs a DOS INT 21H instruction. Before performing the instruction, the function copies the contents of *inregs* and *segregs* to the corresponding registers. The **intdosx** function uses only the DS and ES register values in *segregs*. After the DOS INT instruction returns, **intdosx** copies the current register values to *outregs*, copies the current DS and ES registers to *segregs*, and restores DS. It also copies the status of the system carry flag to the *cflag* field in *outregs*. If this field is nonzero, the system call sets the flag to show an error condition.

The **intdosx** function makes DOS system calls that take an argument in the ES register or that take a DS register value that is different from the default data segment.

The **intdosx** function returns the value of the AX register after the system call is complete. If the *cflag* field in *outregs* is nonzero, an error has occurred. The **intdosx** function sets *_doserrno* to the corresponding error code.

This function is not available under OS/2. For information on calling OS/2 functions from a C program, see "Application Program Interface" in the *IBM Operating System/2 Technical Reference* book.

Examples:

The following example makes a DOS system call to retrieve country-dependent information. Use the data formats for DOS 3.30.

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

int _doserrno
struct country_info {
    int date;
    char currency_sym[5];
    char th_sep[2];
    char dec_sep[2];
    char date_sep[2];
    char time_sep[2];
    char currency_format;
    char currency_sig_dig;
    char time_format;
    unsigned case_map[2];
    char data_list_sep[2];
    unsigned x[5];
    }
   /* Define the three valid date formats. */
char *date_format[3] = ("m d y", "d m y", "y m d");

main()
{
    union REGS inregs, outregs;
    struct SREGS segregs;
    struct country_info far *c;

       /* Reserve space for the table.*/
    c=(struct country_info far *)
       _fmalloc(sizeof(struct country_info));
```

```
      /* Get country-dependent information. */
    inregs.h.ah = 0x38;
    inregs.h.al = 0x00;
    segregs.ds = FP_SEG(c);
    inregs.x.dx = FP_OFF(c);

    intdosx(&inregs, &outregs, &segregs);
    if(outregs.x.cflag) /* Has an error occurred? */
    {
       printf("The DOS error code was %d\n",_doserrno);
       return (1);
    }

    printf("The country code is: %d\n",
     outregs.x.bx);
    printf("The date format is: %s\n",
     date_format[c->date]);
    printf("The currency symbol is: %Fs\n",
     c->currency_sym);
    printf("The thousands separator is: %Fs\n",
     c->th_sep);
    printf("The date separator is: %Fs\n",
     c->date_sep);
    printf("The time separator is: %Fs\n",
     c->time_sep);
    /*   Other values can be printed here. */
}
```

Related Topics:

bdos, intdos, segread, FP_SEG

Note: You can obtain segment values for the *segregs* argument by using either the **segread** function or the **FP_SEG** macro.

Purpose:

Test integer values.

Format:

```
#include <ctype.h>

  /*      Test for alphanumeric  */
  /* ('A'-'Z' or 'a'- 'z', or '0'-'9') */

int isalnum(c)
  /* Test for letter ('A'-'Z' or 'a'-'z') */

int isalpha(c)
  /* Test for ASCII char (0x00-0x7F) */
int isascii(c)
int c;  /* Integer to test */
```

Comments:

The **ctype** routines listed above test a given integer value, returning a nonzero value, if the integer satisfies the test condition, or a zero value, if it does not. These functions assume that the system uses an ASCII character set.

The **isascii** routine produces a meaningful result for all integer values. The remaining routines produce a defined result only for integer values corresponding to the ASCII character set, where **isascii** is true. These routines are also true for the non-ASCII value EOF defined in **stdio.h**.

isalnum - isascii

Example:

The following example analyzes all characters between code 0x0 and code 0x7f, printing A for alphas, AN for alphanumerics, and AS for ASCII characters.

```
#include <stdio.h>
#include <ctype.h>
main()
{
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
   printf("%#04x", ch);
   printf("%3s", isalnum(ch)  ? "AN" : " ");
   printf("%2s", isalpha(ch)  ? "A"  : " ");
   printf("%3s", isascii(ch)  ? "AS" : " ");

   putchar('\n');

   }
}
```

Related Topics:

iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, tolower, toupper

Note: The **ctype** routines are macros.

Purpose:

Tests the handle for a character device.

Format:

```
  /* Required for function declarations */
#include <io.h>

int isatty(handle)
int handle;    /* Handle of device to be tested */
```

Comments:

The **isatty** function determines whether the given *handle* is associated with a character device (a keyboard, screen, printer or serial port).

The **isatty** function returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

Example:

This example tests file handle *fh* and sets long integer *loc* to the current file pointer if *fh* does not correspond to a character device.

```
#include <io.h>

int fh;
long loc;
    .
    .
    .
   /* If not a device, get current position */
if (isatty(fh) == 0)
   loc = tell(fh);
```

Purpose:

Test integer values.

Format:

```
#include <ctype.h>

  /* Test for control char (0x00-0x1f or 0x7f) */
int iscntrl(c)

  /* Test for digit ('0'-'9') */
int isdigit(c)

  /* Test for printable char not including the */
  /* space character(0x21-0x7e) */
int isgraph(c)

  /* Test for lower case ('a'-'z') */
int islower(c)

  /* Test for printable character (0x20-0x7e) */
int isprint(c)

  /* Test for punctuation character (not blank, */
  /* isalnum(c) and iscntrl(c) both false) */
int ispunct(c)

  /* Test for whitespace character */
  /* (0x09-0x0d or 0x20) */
int isspace(c)

  /* Test for upper case ('A'-'Z') */
int isupper(c)

  /* Test for hex digit ('A'-'F', */
  /* 'a'-'f', or '0'-'9') */
int isxdigit(c)

int c;   /* Integer value to test */
```

Comments:

The **ctype** routines listed above test a given integer value, returning a nonzero value, if the integer satisfies the test condition, and zero, if it does not. These tests assume that the system uses an ASCII character set.

These routines produce a defined result only for integer values corresponding to the ASCII character set (only where **isascii** holds true). These routines also have a defined result for the non-ASCII value EOF defined in **stdio.h**.

Example:

The following example analyzes all characters between code 0x0 and code 0x7f, printing **U** for uppercase, **L** for lowercase, **D** for digits, **X** for hexadecimal digits, **S** for spaces, **PU** for punctuation, **PR** for printable characters, **G** for graphics characters, and **C** for control characters. This example prints the code if printable.

The output of this example is a 128-line table showing which of the characters from 0 to 127 possess the attributes tested.

```
#include <stdio.h>
#include <ctype.h>
main()
{
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
   printf(" %c",isprint(ch)  ? ch : '\0');
   printf("%2s",iscntrl(ch)  ? "C"  : " ");
   printf("%2s",isdigit(ch)  ? "D"  : " ");
   printf("%2s",isgraph(ch)  ? "G"  : " ");
   printf("%2s",islower(ch)  ? "L"  : " ");
   printf("%3s",ispunct(ch)  ? "PU" : " ");
   printf("%2s",isspace(ch)  ? "S"  : " ");
   printf("%3s",isprint(ch)  ? "PR" : " ");
   printf("%2s",isupper(ch)  ? "U"  : " ");
   printf("%2s",isxdigit(ch) ? "X"  : " ");

   putchar('\n');
   }
}
```

iscntrl - isxdigit

Related Topics:

isalnum, isalpha, isascii, toascii, tolower, toupper

Note: The **ctype** routines are macros.

Purpose:

Converts an integer value to a character string ending with a null character (\0) and stores the results.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *itoa(value, string, radix)
int value;  /* The number to convert */
char *string;  /* String result */
int radix;  /* Base of value */
```

Comments:

The **itoa** function converts the digits of the given *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value;* it must be in the range 2-36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

The **itoa** function returns a pointer to *string*. There is no error-return value.

Example:

This example converts the decimal value -3445 to an octal number, storing its character representation in the array *buffer*[ ].

```
#include <stdlib.h>

int radix = 8;
char buffer[20];
char *p;
       .
       .
       .
p = itoa(-3445, buffer, radix);  /* p = "171213" */
```

itoa

Related Topics:

itoa, ultoa

Note: The space reserved for *string* must be large enough to hold the returned string. The function can return up to 17 bytes.

Purpose:
Checks keyboard for recent keystrokes.

Format:

```
  /* Required for function declarations */
include <conio.h>

int kbhit( )
```

Comments:
The **kbhit** function checks the keyboard for a recent keystroke.

The **kbhit** function returns a nonzero value if a key has been pressed. Otherwise, it returns zero.

Example:

The following statement tests for the pressing of a key on the keyboard.

If the result is nonzero, a keystroke is waiting in the buffer. You can get it with **getch** or **getche**. If you call **getch** or **getche** without first checking with **kbhit**, the program pauses while waiting for the input of a keystroke.

```
#include <conio.h>

int result;

result = kbhit();
```

labs

Purpose:

Produces the absolute value of a long integer argument.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

long int labs(n)
long int n;   /* Long integer value */
```

Comments:

The **labs** function produces the absolute value of its long integer argument *n*. There is no error-return value. The result is undefined when the argument is the least of the negative long integers (-2147483648), whose absolute value cannot be represented as a long integer.

Example:

This example computes *y* as the absolute value of the long integer -41567.

```
#include <stdlib.h>

long x, y;

x = -41567L;
y = labs(x);    /* y = 41567L */
```

Related Topics:

abs, cabs, fabs

Purpose:

Calculates the value of $x * (2^{exp})$.

Format:

```
#include <math.h>

double ldexp(x, exp)
double x;  /* Floating-point value */
int exp;  /* Integer exponent */
```

Comments:

The **ldexp** function calculates and returns the value of $x * (2^{exp})$. If an overflow results, the function returns +HUGE_VAL or −HUGE_VAL and sets *errno* to ERANGE.

Example:

The following example computes *y* as 1.5 times 2 to the fifth power ($1.5 * 2^5$):

```
#include <math.h>

double x, y;
int p;

x = 1.5;
p = 5;
y = ldexp(x,p);  /* y = 48.0 */
```

Related Topics:

frexp, modf

lfind - lsearch

Purpose:

The **lsearch** and **lfind** functions perform a linear search for the value *key* in an array of elements, each of which is a certain width.

Format:

```
   /* Required only for function declaration */
#include <search.h>

char *lsearch(key,base,num,width,compare)

char *lfind(key,base,num,width,compare)

char *key;    /* Search key */
   /* Pointer to base of search data */
char *base;
   /* Number and width of elements */
unsigned *num,width;
   /* Pointer to compare function */
int (*compare)(const void *element1, const void *element2);
```

Comments:

The **lsearch** and **lfind** functions perform a linear search for the value *key* in an array of *num* elements, each of *width* bytes in size. Unlike **bsearch**, **lsearch** and **lfind** do not require that you sort the array first. The argument *base* is a pointer to the base of the array that is to be searched.

If **lsearch** does not find the *key*, it adds the *key* to the end of the array. If **lfind** does not find the *key*, it does not add the *key* to the array.

The argument *compare* is a pointer to a routine, which you supply, that compares two array elements and returns a value specifying their relationship. Both **lsearch** and **lfind** call the *compare* routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements and then return one of the following values.

Value	Meaning
Not equal to 0	*element1* and *element2* different
0	*element1* identical to *element2*.

Return Value:

If the *key* is found, both **lsearch** and **lfind** return a pointer to that element of the array to which *base* points. If the *key* is not found, **lsearch** returns a pointer to a newly added item at the end of the array, while **lfind** returns NULL.

Example:
This program uses **lfind** function to search for the PATH keyword in the command-line arguments.

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

   /*  Must declare 'compare' as a function */
int compare( );

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

   char **result;
   char *key = "PATH";
   int compare();

/* The following statement finds the *
 * argument that starts with "PATH"  */

   if (result = (char **)lfind((char *)&key,
      (char *)argv, &argc, sizeof(char *),compare))
   printf ("%s found\n",*result);
   else printf ("PATH not found \n");
}

/* The following is a sample 'compare' function */
int compare (arg1, arg2)
char **arg1, **arg2;

{
   return(strncmp(*arg1,*arg2,strlen(*arg1)));
}
```

lfind - lsearch

Related Topics:

bsearch

Purpose:

Converts *time* stored as a long integer to *time* as a structure.

Format:

```
#include <time.h>

struct tm *localtime(time)
const time_t *time;   /* Pointer to stored time */
```

Comments:

The **localtime** function converts a time stored as a **time_t** value to a structure. The **time_t** value *time* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich Mean Time. The **localtime** functions obtain this value from the **time** function.

The function **localtime** breaks down the *time* value, corrects for the local time zone and Daylight Saving Time, if appropriate, and stores the corrected time in a structure of type **tm**. See the **gmtime** function for a description of the fields in a **tm** structure.

DOS does not understand dates prior to 1980. If *time* represents a date before January 1, 1980, **localtime** returns NULL.

The **localtime** function makes corrections for the local time zone if you first set the environment variable TZ. The value of TZ must be a three-letter time zone name such as PST (Pacific Standard Time). A number follows this value, giving the difference between Greenwich Mean Time and the local time zone. If the local time zone is west of the Greenwich meridian, this number is unsigned or has a + sign. If the local time zone is east of the Greenwich meridian, the number has a preceding - sign. A three-letter daylight saving time zone such as PDT (Pacific Daylight Time) can follow this number. The **localtime** function uses the difference between Greenwich Mean Time and local time to adjust the stored time value. If a daylight-saving-time zone is present in the TZ setting, **localtime** also corrects for daylight saving

time. If TZ currently has no value, **localtime** uses the default value EST5EDT.

When you set *TZ*, the system automatically sets three other environment variables, *timezone*, *daylight*, and *tzname*. See the **tzset** function for a description of these variables.

The **localtime** function returns a pointer to the structure result. There is no error return value.

Example:

Suppose that the current local time and date is 3 PM March 31, 1986. The following example reads the system clock and displays the local time in the following message:
the time is Mon Mar 31 15:00:00.00 1986

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

struct tm *newtime;
long ltime;

time(&ltime);
newtime = localtime(&ltime);
printf("the time is %s\n", asctime(newtime));
```

Related Topics:

asctime, ctime, ftime, gmtime, time, tzset

Note: The **gmtime** and **localtime** functions use a single, statically-allocated buffer for the conversion. Each call to one of these functions erases the result of the previous call. The TZ environment variable is an IBM extension and is not part of the ANSI definition of localtime.

Purpose:

Locks or unlocks bytes of a file.

Format:

```
#include <sys\locking.h>
  /* Required for function declarations */
#include <io.h>

int locking(handle, mode, nbyte)
int handle;  /* File handle */
int mode;  /* File locking mode */
long nbyte;  /* Number of bytes to lock */
```

Comments:

The **locking** function locks or unlocks *nbyte* bytes of the file specified by *handle*. Locking bytes in a file prevents subsequent reading and writing of those bytes by other processes. Unlocking a file permits other processes to read or to write to previously locked bytes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbyte* bytes or to the end of the file.

The *mode* variable specifies the action that **locking** is to perform. It must be one of the following.

Mode	Meaning
LK_LOCK	Lock the specified bytes. If the bytes cannot be locked, **locking** tries again after 1 second. If the bytes cannot be locked after 10 attempts, **locking** returns an error.
LK_RLCK	Same as LK_LOCK.
LK_NBLCK	Lock the specified bytes. If bytes cannot be locked, **locking** returns an error.
LK_NBRLCK	Same as LK_NBLCK.

locking

LK_UNLCK Unlock the specified bytes. The bytes must have been previously locked.

You can lock more than one region of a file, but you cannot overlap locked regions. You can unlock no more than one region with a single call.

Note: Refer to the SHARE command in the *IBM Disk Operating System Reference* book.

When unlocking a file, the region of the file unlocked must correspond to a previously locked region. The **locking** function does not unite adjacent regions. If two locked regions are adjacent, you must unlock each region separately.

Remove all locks before closing a file or leaving the program.

The **locking** function returns 0 if it is successful. A return value of −1 indicates failure, and sets *errno* to one of the following values:

Value	Meaning
EACCESS	Locking violation (file already locked or unlocked).
EBADF	Incorrect file handle.
EDEADLOCK	Locking violation. This error returns when you specify the LK_LOCK or LK_RLCK flag, and the file cannot be locked after 10 attempts.
EINVAL	Missing SHARE.COM or SHARE.EXE file.

Example:

The following example tests the DOS version number to tell if the number is at least 3.00. If the number is at least 3.00, this example saves the file pointer position and then locks a region from the beginning of the file to the saved file pointer position. If it succeeds in locking this region, it runs a block of code in which it later unlocks the locked bytes.

```
#include <io.h>
#include <sys\locking.h>
#include <stdlib.h>

extern unsigned char _osmajor;
int fh;
long pos;
       .
       .
       .
if (_osmajor >= 3)   {
   pos = tell(fh);
   lseek(fh, 0L, 0);
   if ( (locking(fh, LK_NBLCK, pos)) != -1) {
                    .
                    .
                    .
      lseek(fh, 0L, 0);
      locking(fh, LK_UNLCK, pos);
      }
   }
```

Related Topics:

creat, open

Note: Under DOS the **locking** function provides file sharing in a network environment. Under OS/2, **locking** provides file sharing for multiple processes.

log - log10

Purpose:

Calculate the natural or base 10 logarithm.

Format:

```
#include <math.h>

  /* Calculate the natural logarithm of x */
double log(x)
  /* Calculate logarithm base10 of x */
double log10(x)
  /* Floating-point value */
double x;
```

Comments:

The **log** function calculates the natural logarithm of *x*. The **log10** function calculates the base 10 logarithm of *x*.

The **log** and **log10** functions return the logarithm result. If *x* is negative, both functions print a DOMAIN error message to the **stderr** data stream, set *errno* to EDOM, and return the value negative HUGE_VAL. If *x* is zero, both functions print a SING error message, return the value negative HUGE_VAL, and set *errno* to ERANGE. For more information about DOMAIN and SING, see the section "Math Errors" in Appendix A, "Error Messages," in this book.

You can change error handling by using the **matherr** routine.

Example:

The following example first calculates the natural logarithm of 1000.0 and then calculates the base 10 logarithm of the same number.

```
#include <math.h>

double x = 1000.0, y;

   /* The natural logarithm, y = 6.907755 */
y = log(x);
   /* The base 10 logarithm, y = 3.0 */
y = log10(x);
```

Related Topics:

exp, matherr, pow

longjmp

Purpose:

Restores a stack environment that **setjmp** previously saved.

Format:

```
#include <setjmp.h>

void longjmp(env, value)
  /* Variable in which to store environment */
jmp_buf env;
  /* Value to be returned to the setjmp call */
int value;
```

Comments:

The **longjmp** function restores a stack environment previously saved in *env* by **setjmp**. The **setjmp** and **longjmp** functions provide a way to perform a nonlocal **goto**. Use these functions to pass program control to error-handling or recovery code in a previously-called function 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. Performance of the program resumes as if the **setjmp** call had just returned the given *value*. All variables, except register variables, that are accessible to the function that receives control contain the values they had when you called **longjmp**. The values of register variables are unpredictable.

Note: You must call the **longjmp** function before the function that called **setjmp** returns. Calling **longjmp** after the function calling **setjmp** returns causes unpredictable program behavior. You may call **longjmp** from within a signal-handling routine that you defined, but if a signal occurs during the call to **longjmp**, the results are unpredictable.

The *value* returned by **longjmp** must be nonzero. If you give a zero argument for *value*, **longjmp** substitutes a 1 in the return.

The **longjmp** function does not return a value.

Example:

The following example saves the stack environment at the statement:

```
if(setjmp(mark) != 0) ...
```

On the first performance of the **if** condition, the system saves the environment in *mark*. The condition in the **if** statement is false because the **setjmp** function returns 0 when saving the environment. The system prints the message **setjmp has been called**.

The subsequent call to function **p** tests for a local error condition that might cause the program to perform **longjmp**. Then control passes to the original **setjmp**, using the environment saved in *mark*. This time the condition is true. The **longjmp** function returns a value of -1. The system prints **longjmp has been called**. It then runs the **recover** function, which you supply, and exits.

longjmp

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

jmp_buf mark;
main()
{
   if (setjmp(mark) != 0) {
      printf("longjmp has been called\n");
      recover();
      exit(1);
      }
   printf("setjmp has been called\n");
                        .
                        .
                        .
   p();
                        .
                        .
                        .
}
p()
{
   int error = 0;
                        .
                        .
                        .
   if (error != 0)
        longjmp(mark, -1);
                        .
                        .
                        .
}
recover()
{
   /* Ensure exiting will not corrupt data files */
                        .
                        .
                        .
}
```

Related Topics:

setjmp

Note: The values of register variables in the function calling **setjmp** might not be restored to the proper values after a **longjmp** runs.

Purpose:

Moves the file pointer to a new location.

Format:

```
  /* Required for function declarations */
#include <io.h>

long lseek(handle, offset, origin)
  /* Handle referring to open file */
int handle;
  /* Number of bytes from origin */
long offset;
int origin;   /* Initial position */
```

Comments:

The **lseek** function moves any file pointer associated with *handle* to a new location that is *offset* bytes from the *origin*. The next operation on the file takes place at the new location. *Origin* must be one of the following constants, defined in **stdio.h**:

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file.

The **lseek** function can 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.

lseek

The **lseek** function returns the offset, in bytes, of the new position from the beginning of the file. A return value of −1L indicates an error, and *errno* is set to one of the following values:

Value	Meaning
EBADF	The file handle is incorrect.
EINVAL	The value for *origin* is incorrect, or the position specified by *offset* is before the beginning of the file.

On devices incapable of seeking (such as keyboards and printers), the return value is undefined.

Example:

The following example shows two calls to **lseek**. After opening the DATA file for reading, the program tries to move the file pointer to the beginning of the file. It prints:

```
lseek to beginning failed
```

if this operation is unsuccessful. Later the program calls **lseek** with an origin of 1 to get the position five bytes beyond the current file pointer.

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

int fh;
long position;

fh = open("data", O_RDONLY);
                .
                .
                .
   /* 0 offset from beginning */

position = lseek(fh, 0L, SEEK_SET);
if (position == -1L)
   perror("lseek to beginning failed");
                .
                .
                .
  /* Move from current position */
position = lseek(fh, 5L, SEEK_CUR);
if (position == -1L)
   perror("lseek 5 beyond current position failed");
                .
                .
                .
/* Go to end of file */
position = lseek(fh, 0L, SEEK_END);
if (position == -1L)
   perror("lseek to end failed");
```

Related Topics:

fseek, tell

ltoa

Purpose:

Converts digits to a null-ended character string and stores the result.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char *ltoa(value, string, radix)
long value;    /* Number to convert */
char *string;    /* String result */
int radix;    /* Base of value */
```

Comments:

The **ltoa** function converts the digits of the given long *value* to a null−ended character string and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2−36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (−).

The **ltoa** function returns a pointer to *string*. There is no error return.

Example:
This example converts the long integer -344155 to the ASCII string "-344155".

```
#include <stdlib.h>

int radix = 10;
char buffer[20];
char *p;

p = ltoa(-344155L, buffer, radix);
     /* p = "-344155" */
```

Related Topics:

itoa, ultoa

Note: The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

malloc

Purpose:

Reserves a block of storage.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>
void *malloc(size)
   /* Bytes in reserved block of storage */
size_t size;
```

Comments:

The **malloc** function reserves a block of storage of at least *size* bytes. (The block might be larger than *size* bytes due to space required for alignment and for maintenance information.)

The **malloc** function returns a pointer to the reserved space. The storage space to which the return value points is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage available.

Example:

The following example reserves space for 20 integers.

```
#include <stdlib.h>

int *intarray;

intarray = (int *)malloc(20*sizeof(int));
```

Related Topics:

calloc, free, realloc, _fmalloc, _nmalloc

Note: The call **malloc**(0) does not return NULL but, instead, reserves a 0-length item (header only) in the heap. The resulting pointer can be passed to **realloc** to adjust the *size* at any time. In the small and medium models, a call to **malloc** is equivalent to a call to **_nmalloc**. In the compact and large models, a call to **malloc** is equivalent to a call to **_fmalloc**.

matherr

Purpose:

Processes errors produced by math library functions.

Format:

```
#include <math.h>

int matherr(x)
   /* Math exception information */
struct exception *x;
```

Comments:

The **matherr** function processes errors generated by the functions of the math library. The math functions call **matherr** whenever they detect an error. You can provide a different definition of the **matherr** function to carry out special error-handling.

When an error occurs in a math routine, **matherr** is called with a pointer to the following structure (defined in **math.h**) as an argument:

```
struct exception {
  int type;
  char *name;
  double arg1, arg2, retval;
  };
```

The *type* variable 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
UNDERFLOW	Underflow range error
TLOSS	Total loss of significance
PLOSS	Partial loss of significance.

The *name* is a pointer to a null-ended string containing the name of the function that caused the error. The *arg1* and *arg2* variables specify the argument values that caused the error. (If only one argument is given, it is stored in *arg1*).

The *retval* is the default return value for this error; you can change the return value. The return value from **matherr** must specify whether or not an error actually occurred. If **matherr** returns zero, an error message appears, and *errno* is set to an appropriate error value. If **matherr** returns a nonzero value, no error message appears and *errno* remains unchanged.

The **matherr** routine should return zero to indicate an error and nonzero to indicate successful corrective action.

Example:

The following example is a definition that you can create to handle errors from the **log** or **log10** functions. The arguments to these logarithmic functions must be positive **double** values. This routine processes a negative value in an argument (a DOMAIN error) by returning the log of its absolute value. It suppresses the error message normally displayed when this error occurs. If the error is a zero argument, or if some other routine produced the error, the example takes the default actions.

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

int matherr(x)
struct exception *x;
{
   if (x->type == DOMAIN) {
      if (strcmp(x->name, "log") == 0) {
         x->retval = log(-(x->arg1));
         return(1);
         }
      else if (strcmp(x->name, "log10") == 0) {
         x->retval = log10(-(x->arg1));
         return(1);
         }
      }
   return(0); /* Use default actions */
}
```

Related Topics:

acos, asin, atan, atan2, bessel, cabs, cos, cosh, exp, hypot, log, pow, sin, sinh, sqrt, tan

Purpose:

The **_memavl** function returns the approximate size in bytes of the storage space available for dynamic allocation in the default data segment.

Format:

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

size_t _memavl( )
```

Comments:

The **_memavl** function returns the approximate size in bytes of the storage space available for dynamic allocation in the default data segment. You can use this function with **calloc**, **malloc**, or **realloc** in the small- and medium-storage models and with **_nmalloc** in all storage models.

The **_memavl** function returns the size in bytes as an unsigned integer.

_memavl

Example:

```
#include <malloc.h>
#include <stdio.h>
main()
   {
   long *longptr;

   printf("Memory available before"
       "malloc = %u\n", _memavl( ));
   longptr = (long*)malloc(5000*sizeof(long));
   printf("Memory available after"
       "malloc = %u\n", _memavl( ));
   }
```

Output:

(Actual numbers may vary slightly.)

```
Memory available before malloc = 61293
Memory available after malloc = 40959
```

Related Topics:

calloc, **malloc**, **_freect**, **realloc**, and **stackavail**

Purpose:

Copies bytes of *src* to *dest*.

Format:

```
   /* Required for function declarations */
#include <memory.h>
   /* Use either string.h or memory.h */
#include <string.h>

void *memccpy(dest, src, c, cnt)
void *dest;  /* Pointer to destination */
void *src;   /* Pointer to source */
int c;   /* Last character to copy */
unsigned cnt;  /* Number of characters */
```

Comments:

The **memccpy** function copies zero or more bytes of *src* to *dest*. It copies up to and including the first occurrence of the character *c* or until *cnt* bytes have been copied, whichever comes first.

If the character *c* is copied, **memccpy** returns a pointer to the byte in *dest* that immediately follows the character. If *c* is not copied, **memccpy** returns NULL.

memccpy

Example:

The following example copies up to 100 bytes from the source to a buffer until it copies the '\n' character:

```
#include <memory.h>

char buffer[100], source[100];
char *result;
        .
        .
        .

result = memccpy(buffer, source, '\n', 100);
```

Related Topics:

memchr, memcmp, memcpy, memset

Purpose:

Searches *buf* for the first occurrence of *c*.

Format:

```
  /* Required for function declarations */
#include <string.h>
void *memchr(buf, c, cnt)
const void *buf; /* Pointer to buffer */
int c; /* Character for which to search */
size_t cnt; /* Number of characters */
```

Comments:

The **memchr** function searches the first *cnt* bytes of *buf* for the first occurrence of *c* converted to a character. The search continues until it finds *c* or examines *cnt* bytes.

The **memchr** function returns a pointer to the location of *c* in *buf*. It returns NULL if *c* is not within the first *cnt* bytes of *buf*.

Example:

The following example finds the first occurrence of 'a' in the buffer. If 'a' is not in the first 100 bytes, **memchr** returns a NULL.

```
#include <string.h>

char buffer[100];
char *result;
        .
        .
        .

result = memchr(buffer, 'a', 100);
```

Related Topics:

memccpy, memcmp, memcpy, memset

memcmp

Purpose:

Compares *buf1* and *buf2*.

Format:

```
   /* Required for function declarations */
#include <string.h>
int memcmp(buf1, buf2, cnt)
const void *buf1;  /* First buffer */
const void *buf2;  /* Second buffer */
size_t cnt;  /* Number of characters */
```

Comments:

The **memcmp** function compares the first *cnt* bytes of *buf1* and *buf2* and returns a value indicating their relationship, as follows:

Value	Meaning
Less than 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
Greater than 0	*buf1* greater than *buf2*.

The **memcmp** function returns an integer value as described above.

Example:

The following example compares *first*[ ] and *second*[ ] to see which, if either, is greater. If the first 100 bytes are the same, then the **memcmp** function considers them equal.

```
#include <string.h>

char first[100], second[100];
int result;
       .
       .
       .
result = memcmp(first, second, 100);
```

Related Topics:

memccpy, memchr, memcpy, memset

memcpy

Purpose:

Copies bytes of *src* to *dest*.

Format:

```
   /* Required for function declarations */
#include <memory.h>

char *memcpy(dest, src, cnt)
void *dest;  /* Pointer to destination */
const void *src;  /* Pointer to source */
size_t cnt;  /* Number of characters */
```

Comments:

The **memcpy** function copies *cnt* bytes of *src* to *dest*. If some regions of *src* and *dest* overlap, **memcpy** ensures that the original *src* bytes in the overlapping region are copied before writing over them.

The **memcpy** returns a pointer to *dest*.

Example:

The following example moves 200 bytes from source to destination, and returns a pointer to destination.

```
#include <memory.h>

char source[200], destination[200];
            .
            .
            .
memcpy(destination, source, 200);
```

Related Topics:

memccpy, memchr, memcmp, memset

Purpose:

Compares bytes in buffers without regard to the case of the letters.

Format:

```
  /*Required for function declarations*/
#include <memory.h>

int memicmp (buf1, buf2, cnt)
void *buf1;  /* First buffer */
void *buf2;  /* Second buffer */
unsigned int cnt;   /* Number of characters */
```

Comments:

The **memicmp** function compares the first *cnt* bytes of *buf1* and *buf2* without regard to the case of letters in the two buffers. Uppercase and lowercase letters are considered equivalent. The **memicmp** function returns a value indicating the relationship of *buf1* and *buf2* as follows.

Value	Meaning
Less than 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
Greater than 0	*buf1* greater than *buf2*.

The **memicmp** function returns an integer value.

memicmp

Example:

The following example copies two strings that each contain a substring of 29 characters that is the same except for its case. The example then compares the first 29 bytes without regard to case.

```
#include <memory.h>

char first[100], second[100];
int result;

strcpy(first,"Those Who Will Not Learn"
   "From History");
strcpy(second,"THOSE WHO WILL NOT LEARN"
   "FROM their mistakes");
result = memicmp(first,second,29);
printf("%d\n",result);
```

Result:

```
0
```

Related Topics:

memccpy, memchr, memcmp, memcpy, memset

Purpose:

Sets first *cnt* bytes of *dest* to character *c*.

Format:

```
   /* Required for function declarations */
#include <string.h>

void *memset(dest, c, cnt)
void *dest;  /* Pointer to destination */
int c;  /* Character to set */
size_t cnt;  /* Number of characters */
```

Comments:

The **memset** function sets the first *cnt* bytes of *dest* to the value *c*, converted to a character.

The **memset** function returns a pointer to *dest*.

Example:
The following example sets the first 100 bytes of the buffer to NULL.

```
#include <string.h>

char buffer[100];

memset(buffer, '\0', 100);
```

Related Topics:

memccpy, memchr, memcmp, memcpy

mkdir

Purpose:

Creates a new directory.

Format:

```
   /* Required for function declarations */
#include <direct.h>
int mkdir(pathname)
   /* Path name for new directory */
char *pathname;
```

Comments:

The **mkdir** function creates a new directory with the specified *pathname*. Only one directory can be created at a time, so only the last component of *pathname* can name a new directory.

The **mkdir** function returns the value 0 if the a directory was created. A return value of −1 shows an error, and *errno* is set to one of the following values.

Value	Meaning
EACCES	The directory was not created: the given name is the name of an existing file, directory, or device.
ENOENT	The *pathname* was not found.

Example:

The following example creates two new directories: one at the root on drive **B:**, and one in the **tmp** subdirectory of the current working directory.

```
#include <direct.h>

int result;

result = mkdir("b:\\aleng");
             .
             .
             .
result = mkdir("tmp\\aleng");
```

Related Topics:

chdir, rmdir

mktemp

Purpose:

Creates a unique filename from *template*.

Format:

```
   /* Required for function declarations */
#include <io.h>

char *mktemp(template)
char *template;   /* Filename pattern */
```

Comments:

The **mktemp** function creates a unique filename by changing the given *template*. The *template* argument has the form:

basexxxxxx

where *base* is the part of the new filename supplied by the user and the *x*s are placeholders for the part supplied by **mktemp**. The **mktemp** function preserves *base* and replaces the six trailing *x*s with an alphanumeric character followed by a 5-digit value. The 5-digit value is a unique number identifying the calling process. The alphanumeric character is zero the first time **mktemp** is called with a *template*.

In subsequent calls from the same process with the same *template*, **mktemp** checks to see whether 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 the lowercase letters 'a' to 'z'.

The **mktemp** function returns a pointer to the modified *template*. The return value is NULL if the *template* argument has a syntax error or no more unique names can be created from the *template*.

Example:

The following example calls **mktemp** to produce a unique filename.

```
#include <io.h>

char *template = "fnXXXXXX";
char *result;

result = mktemp(template);
```

Related Topics:

fopen, getpid, open

Note: The **mktemp** function produces unique filenames but does not create or open files.

modf

Purpose:

Breaks down floating-point values into fractional and integer parts.

Format:

#include <math.h>

double modf(*x,intptr*)
  /* Floating—point value */
double *x*;
  /* Pointer to stored integer portion */
double **intptr*;

Comments:

The **modf** function breaks down the floating—point value *x* into fractional and integer parts. The signed fractional portion of *x* is returned. The integer portion is stored as a floating—point value at *intptr*.

The **modf** function returns the signed fractional portion of *x*. There is no error return.

Example:

The following example breaks the floating-point number -14.87654321 into its fractional and integer components:

```
#include <math.h>

double x, y, n;

x = -14.87654321;
y = modf(x, &n);  /* y = -0.87654321, n = -14.0 */
```

Related Topics:

frexp, ldexp

Purpose:

Copies *nbytes* bytes from a specified source address to a specified destination address.

Format:

```
   /* Required only for function declarations */
#include <memory.h>

void movedata(srcseg,srcoff,destseg,destoff,nbytes)
   /* Segment address of source */
unsigned int srcseg;
   /* Offset value of source */
unsigned int srcoff;
   /* Segment address of destination */
unsigned int destseg;
   /* Offset value of destination */
unsigned int destoff;
   /* Number of bytes to copy */
unsigned nbytes;
```

Comments:

The **movedata** function copies *nbytes* of data from the source address specified by *srcseg*:*srcoff* to the destination address specified by *destseg*:*destoff*.

The **movedata** function moves far data in small or medium-model programs where segment addresses of data are not implicitly known. In large-model programs, use the **memcpy** function because segment addresses are implicitly known.

Under OS/2, references to segments are translated into selector values.

movedata

Note:

Segment values for the *srcseg* and *destseg* arguments can be obtained by using either the **segread** function or the FP_SEG macro.

The **movedata** function does not handle all cases of overlapping moves correctly. *Overlapping moves* occur when part of the destination is the same storage area as part of the source. Overlapping moves are handled correctly in the **memcpy** function.

Example:

The following example moves 512 bytes of data from *src* to *dest*.

```
#include <memory.h>
#include <dos.h>

char far *src;
char far *dest;
main()
{

movedata(FP_SEG(src),FP_OFF(src),
    FP_SEG(dest), FP_OFF(dest), 512);
}
```

Related Topics:

memcpy, **segread**, FP_SEG

Purpose:

Returns the size in bytes of the storage block that C reserved during a call to a **calloc**, **malloc**, or **realloc** function.

Format:

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

size_t _msize(ptr)
void *ptr;  /* Pointer to memory block */
```

Comments:

The **_msize** function returns the size in bytes of the storage block reserved by a call to **calloc**, **malloc**, or **realloc**.

The **_msize** function returns the size in bytes as an unsigned integer.

Example:

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

main()

   {
   long *oldptr;
   size_t newsize = 64000;

   oldptr = (long *)malloc(10000*sizeof(long));
   printf("Size of block of storage pointed to by
           oldptr = %u\n", _msize(oldptr));

   if(_expand(oldptr,newsize) != NULL)
      printf("expand was able to increase block to
      %u\n", _msize(oldptr));
   else
      printf("expand was able to increase block to
      only %u\n", _msize(oldptr));
   }
```

_msize

Output:

Size of block of storage pointed to by oldptr = 40000
expand was able to increase block to only 44836.

(Actual numbers may vary slightly.)

Related Topics:

calloc, _expand, malloc, realloc

Purpose:

Frees a block of storage.

Format:

```
 /* Required only for function declarations */
#include <malloc.h>
void _nfree(ptr)
void near *ptr;
```

Comments:

The **_nfree** function frees a block of storage. The argument *ptr* points to a block of storage previously reserved through a call to **_nmalloc**. The number of bytes freed is the number of bytes specified when you reserved the block. After the call, the freed block is again available to be reserved. If *ptr* is NULL, **_nfree** ignores it.

Note: Attempting to free a pointer not reserved with **_nmalloc** can affect subsequent allocation and cause errors.

Example:

The following example reserves 100 bytes and then frees them.

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

void near *alloc;

/* Test for a valid pointer */
if ((alloc = _nmalloc(100)) == NULL)
   printf("unable to reserve storage\n");
else {
/* Free storage for the heap */
     _nfree(alloc);
     }
```

Related Topics:

_nmalloc, free, malloc

_nmalloc

Purpose:

Reserves a storage block.

Format:

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

void near *_nmalloc(size)
size_t size; /* Bytes in allocated block */
```

Comments:

The **_nmalloc** function reserves a storage block of at least *size* bytes inside the default data segment. The block can be larger than *size* bytes because of the space required for aligning the block.

The **_nmalloc** function returns a near pointer. The storage space to which the return value points is guaranteed to be aligned for storage of any type of object. To get the pointer to a type, you must use a type cast on the return value. The return value is NULL if there is not enough storage available.

Example:

The following example reserves space for 20 integers.

```
#include <malloc.h>

int near *array;

array = (int *)_nmalloc(20*sizeof(int));
```

Related Topics:

_nfree, _nmsize, malloc, realloc

Purpose:

Returns the size in bytes of the storage block reserved by a call to **_nmalloc**.

Format:

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

size_t _nmsize(ptr)
void near *ptr;  /* Pointer to memory block */
```

Comments:

The **_nmsize** function returns the size in bytes of the storage block that C reserves during a call to the **_nmalloc** function. The **_nmsize** function returns the size in bytes as an unsigned integer.

Example:

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

main()
   {
   char near *stringarray;

   stringarray = _nmalloc(200*sizeof(char));
   if (stringarray != NULL)
      printf("%u bytes allocated\n",
     _nmsize(stringarray));
   else
      printf("Allocation request failed.\n");
   }
```

Related Topics:

_ffree, _fmalloc, _fmsize, malloc, _msize, _nfree, _nmalloc

onexit

Purpose:

Receives the address of a function to call when the program ends normally.

Format:

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

onexit_t  onexit (func)
onexit_t func;
```

Comments:

The **onexit** function receives the address of a function *func* to call when the program ends normally. Successive calls to **onexit** create a register of functions that run last-in, first-out. You can place no more than 32 functions in the register with calls to **onexit**. If you exceed 32 functions, **onexit** returns the value NULL. The functions passed to **onexit** cannot take parameters.

If successful, **onexit** returns a pointer to the function, otherwise it returns a NULL value.

Example:

This example specifies and defines four distinct functions that run consecutively at the completion of **main**.

```
#include <stdlib.h>
main()
   {
   int fn1(), fn2(), fn3(), fn4();

   onexit(fn1);
   onexit(fn2);
   onexit(fn3);
   onexit(fn4);
   printf("This is run first.\n");
   }

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

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

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

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

open

Purpose:

Opens a file for reading or writing.

Format:

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

int open(pathname, oflag[, pmode])
char *pathname;          /* File pathname */
int oflag;    /* Type of operations allowed */
int pmode;          /* Permission setting */
```

Comments:

The **open** function opens the file specified by *pathname* and prepares the file for subsequent reading or writing as defined by *oflag*. The *oflag* is an integer expression formed by combining one or more of the following manifest constants, defined in **fcntl.h**. When more than one manifest constant is given, the constants are joined with the bitwise OR operator |.

Oflag	Meaning
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file. This has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.

O_RDWR	Open the file for both reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TRUNC	Open and truncate an existing file to 0 length. The file must have write permission. The contents of the file are destroyed.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.
O_BINARY	Open the file in binary (untranslated) mode. (See **fopen** for a description of binary mode.)
O_TEXT	Open the file in text (translated) mode. (See **fopen** for a description of text mode.)

You must specify one of the access mode flags, O_RDONLY, O_WRONLY, or O_RDWR. There is no default.

Note: When you open files in text mode for reading only, Ctrl + Z is interpreted as an end-of-file character. When you open files in text mode for reading only or for writing and reading, **open** attempts to remove any Ctrl + Z characters from the end of the file when it opens the file.

CAUTION:
O_TRUNC destroys the complete contents of an existing file. Use it with care.

The *pmode* argument is required only when O_CREAT is specified.

If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the permission settings for the file, 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, all files are readable; it is not possible to give write–only permission. 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** in this chapter.)

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
EACCESS	The given *pathname* is a directory; or the file is read–only but an open for writing was attempted; or a sharing violation occurred. (The sharing mode of the file does not allow the specified operations unless you have PC DOS Version 3.00 or later).
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available. There are too many open files.
ENOENT	File or *pathname* not found.

Example:

This example tries to open files DATA1 and DATA2 for writing. It prints an **open failed** message whenever the **open** function returns an error value.

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

int fh1, fh2;
main ()
{
fh1 = open("data1", O_RDONLY);
if (fh1 == -1)
   perror("open failed on input file");

fh2 = open("data2", O_WRONLY | O_TRUNC | O_CREAT,
S_IREAD | S_IWRITE);

if (fh2 == -1)
   perror("open failed on output file");
}
```

Related Topics:

access, chmod, close, creat, dup, dup2, fopen, sopen, umask

outp

Purpose:

Writes specified values to an output port.

Format:

```
 /* Required for function declarations */
#include <conio.h>

int outp(port, value)
unsigned port;  /* Port number */
int value;  /* Output value */
```

Comments:

The **outp** function writes the specified *value* to the output port specified by *port*. The *port* argument can be any unsigned integer in the range of 0 through 65535; *value* can be any integer in the range of 0 through 255.

The **outp** function returns *value*. There is no error return.

Example:

The following example sends a byte to the port to which *port* is currently set.

```
#include <conio.h>

int byte_val;
unsigned port;
  .
  .
  .
outp(port, byte_val);
```

Related Topics:

inp

Note: Under OS/2, **outp** uses privileged **OUT** instructions that require you to set up your system to get access to the input/output privilege level (IOPL) and to provide a **.DEF** file for your program. IBM recommends **outp** only for getting access to ports for graphics adapters. For more information on IOPL, see the *IBM Operating System/2 Technical Reference* book.

perror

Purpose:

Prints an error message to **stderr**.

Format:

```
   /* Required for function declarations */
#include <stdio.h>

void perror(string)
const char *string;  /* User supplied message */

int errno;  /* Error number */
int sys_nerr;  /* Number of system messages */
  /* Array of error messages */
char *sys_errlist [sys_nerr];
```

Comments:

The **perror** function prints an error message to **stderr**. The *string* argument is printed first, followed by a colon, the system error message for the last library call that produced an error, and a newline character. If *string* is a NULL pointer or a pointer to a NULL string, **perror** prints only the system error message.

The error number is stored in the variable *errno*, which you declare at the external level. The **perror** function gets access to the system error messages through the variable *sys_errlist*, which is an array of messages arranged 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, subsequent calls might write over the *errno* value.

The **perror** function returns no value.

Example:

The following example tries to open a stream. If the **open** function fails, the example prints a message and ends the program.

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

int fh

if ((fh = open("data",O_RDONLY)) == -1) {
   perror("Could not open data file");
   abort();
   }
```

Related Topics:

clearerr, ferror

Note: Under DOS, some of the *errno* values listed in **errno.h** are not used. See Appendix A, "Error Messages," in this book for a list of *errno* values used on DOS and the corresponding error messages. The **perror** function prints an empty string for any *errno* value not used under DOS.

pow

Purpose:

Computes x^y.

Format:

```
#include <math.h>
double pow(x,y)
double x;  /* Number to be raised */
double y;  /* Power of x */
```

Comments:

The **pow** function computes *x* raised to the *y*th power.

The **pow** function returns the value of x^y. If *y* is zero, **pow** returns the value 1. If *x* is zero and *y* is negative, **pow** sets *errno* to EDOM and returns 0. If both *x* and *y* are 0, or if *x* is negative and *y* is not an integer, **pow** prints a DOMAIN error message to **stderr**, sets *errno* to EDOM, and returns 0. If an overflow results, the function sets *errno* to ERANGE and returns either positive or negative HUGE_VAL. No message is printed for overflow or underflow conditions. The **pow** function does not recognize integral floating-point values greater that 2^{64}.

Example:

The following example calculates the value of 2^3:

```
#include <math.h>

double x = 2.0, y = 3.0, z;
      .
      .
      .
z = pow(x,y);    /* z = 8.0 */
```

Related Topics:

exp, log, sqrt

Purpose:

Formats and prints characters to **stdout**.

Format:

#include <stdio.h>

int printf(*format-string*[, *argument...*])
const char **format-string*; /* Format-control string */

Comments:

The **printf** function formats and prints a series of characters and values to the standard output stream **stdout**. The *format-string* consists of ordinary characters, escape sequences, and format specifications. The ordinary characters are copied in order of their appearance to **stdout**. Format specifications, beginning with a percent sign (%), determine the output format for any *arguments* following the *format-string*.

The *format-string* is read left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and put out according to the format specification. The second format specification causes the second *argument* to be converted and put out, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications. A format specification has the following form:

%[*flags*] [*width*] [.*precision*] [F|N|h|l|L]*type*

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, "%s").

The following optional fields control other aspects of the formatting.

Field	Description
flags	Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes.
width	Minimum number of characters output.
precision	Maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values.
F,N	Prefixes that let you ignore the default addressing conventions of the storage model that you are using: **F** In a small model, prints a value previously declared **far**. **N** In medium, large, and huge models, prints values previously declared **near**. Use **F** and **N** only with the **s** and **p** *type* characters because they are relevant only with arguments that pass a pointer. F and N are IBM extensions.
h,l,L	Size of argument expected: **h** A prefix with the integer types **d**, **i**, **o**, **u**, **x**, and **X** that specifies that the argument is **short int**. **l** A prefix with **d**, **i**, **o**, **u**, **x**, and **X** types that specifies that the argument is a **long int**; with the **e**, **E**, **f**, **g**, or **G** types, it shows that the argument is **double** instead of **float**. **L** A prefix with **e**, **E**, **f**, **g**, or **G** types that specifies that the argument is **long double**.

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 field, the character is simply copied to STDOUT. For example, to print a percent sign character, use "%%".

The *type* characters and their meanings are given in the following table.

Character	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using "abcdef".
X	Integer	Unsigned hexadecimal integer, using "ABCDEF".
f	Floating-point	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	Floating-point	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	Floating-point	Identical to the "e" format except that "E" introduces the exponent instead of "e".

Character	Argument	Output Format
g	Floating-point	Signed value printed in "f" or "e" format, whichever is more compact for the given value and *precision* (see below). The "e" format is used only when the exponent of the value is less than −4 or greater than *precision*. Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
G	Floating-point	Identical to the "g" format except that "E" introduces the exponent (where appropriate) instead of "e".
c	Character	Single character.
s	Pointer to char	Characters printed up to the first null character ('\0') or until *precision* 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, *yyyy* is the offset, and the digits *x* and *y* are uppercase hexadecimal digits. **%Np** prints only the offset of the address *yyyy*. Since **%p** expects a pointer to a **far** value, pointer arguments to **p** must be cast to **far** in small model programs.

The *flag* characters and their meanings are as follows (notice that more than one *flag* can appear in a format specification):

Flag	Meaning	Default
–	Left–justify the result within the field *width*.	Right–justify.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The "+" flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the **o**, **x**, or **X** formats, the "#" flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix
#	When used with the **f**, **e**, or **E** formats, 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** formats, the "#" flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Ignored when used with **c**, **d**, **i**, **u**, or **s**.	Decimal point appears only if digits follow it; trailing zeros are truncated.

Width is a nonnegative 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 on the left or the right (depending on whether the "–" flag is specified) until the minimum width is reached. If *width* is prefixed with a zero (0), zeros are added until the minimum width is reached (not useful for left-justified numbers).

Width 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 can be an asterisk (*), in which case an argument from the argument-list supplies the value. The *width* argument must precede the value being formatted in the argument list.

Precision is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

The *precision* specification may be an asterisk (*), in which case an 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 the *precision* is omitted depend upon the *type*, as shown in the following table.

Type	Meaning	Default
i d u o x X	*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	*Precision* specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.	Default *precision* is six. If *precision* is zero or the period appears without a number following it,no decimal point is printed.

Type	Meaning	Default
f	*Precision* 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 zero; if *precision* is explicitly zero, no decimal point is printed.
g G	*Precision* specifies the maximum number of significant digits printed.	All significant digits are printed.
c	No effect.	Character printed.
s	*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.

The **printf** function returns the number of characters printed.

Representation of special floating-point values:

If you call **printf** with a floating point format specifier and the corresponding argument is infinite, indefinite, or not-a-number, the output from **printf** appears as follows:

VALUE	OUTPUT
+ **infinity**	1.#**INF***random digits*
- **infinity**	-1.#**INF***random digits*
indefinite	*digit*.#**IND***random digits*
not-a-number	*digit*.#**NaN***random digits*

where:

digit and *random digits*
are unpredictable values.

printf

Example:

The following example prints data in a variety of formats:

```
#include <stdio.h>

main()
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    int *ptr;
    double fp = 251.7366;

         /* Output integer in various formats  */
    printf("%d   %+d  %06d   %X   %x   %o\n\n",
       count, count, count, count, count, count);

         /* Show how %n gets string length     */
    printf("1234567890123%n4567890123456789O\n\n",
           &count);
    printf("Value of count should be 13; "
           "count = %d\n\n", count);

         /* Show character in two formats      */
    printf("%10c%5c\n\n",ch,ch);

        /* Show string in two formats         */
    printf("%25s\n%25.4s\n\n",string, string);

      /* Show floating point in 4 formats.  */
    printf("%f    %.2f    %e    %E\n\n",
            fp, fp, fp, fp);

    /* Show output of different bases     */
    printf("%i    %i    %i\n\n", hex, oct, dec);

      /* Show output using pointers         */
    ptr = &count;
    printf("%Np    %p    %Fp\n",
           ptr, (int far *)ptr, (int far *)ptr);
}
```

Output:

(Actual numbers may vary slightly.)

```
234   +234  000234   EA   ea   352

12345678901234567890123456789O

Value of count should be 13; count = 13

        h    h

               computer
                   comp

251.736600    251.74    2.517366e+002    2.517366E+002

16    8    10

12FA    41E4:12FA    41E4:12FA
```

Related Topics:

fprintf, scanf, sprintf

putc - putchar

Purpose:

Writes a character to the output stream.

Format:

```
#include <stdio.h>

  /* Write a character to stream */
int putc(c, stream)
int c;   /* Character to write */
FILE *stream;  /* Pointer to file structure */

int putchar(c)  /* Write a character to stdout */
int c;    /* Character to write */
```

Comments:

The **putc** function writes the single character *c* to the output *stream* at the current position. The **putchar** function is identical to **putc**(*c*, **stdout**).

The **putc** and **putchar** functions return the character written. A return value of EOF indicates an error which could be caused by an attempt to write to a read-only file, specifying a non-valid stream pointer, or no space left on the device. Because the EOF value is a legitimate integer value, use the **ferror** function to verify that an error occurred.

Example:

The following example writes the contents of a buffer to a data stream. In this example, the body of the **for** statement is null because the example carries out the writing operation in the test expression.

```
#include <stdio.h>

FILE *stream;
char buffer[81]="strings";
int i, ch;
main()
{
for (i = 0; (i < 81) && ((ch = putc(buffer [i],
     stream)) != EOF); i++);
}
```

Related Topics:

fputc, **fputchar**, **getc**, **getchar**

Note: The **putc** and **putchar** functions are identical to **fputc** and **fputchar** but are macros, not functions.

putch

Purpose:

Writes the character *c* directly to the screen.

Format:

```
 /* Required for function declarations */
#include <conio.h>

int putch(c)
int c;  /* Character to put out */
```

Comments:

The **putch** function writes the character *c* directly to the screen. If the action is successful, **putch** returns *c*. In case of error in OS/2 mode, **putch** returns EOF.

Example:

The following example shows how to define the **getche** function using **putch** and **getch**.

```
#include <conio.h>

int getche()
{
   int ch;

   ch = getch();
   putch(ch);
   return(ch);
}
```

Related Topics:

cprintf, getch, getche

Purpose:

Adds or modifies environment variables.

Format:

```
 /* Required for function declarations */
#include <stdlib.h>

int putenv(envstring)
 /* Environment string definitions */
char *envstring;
```

Comments:

The **putenv** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process runs (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 value of the variable. If *varname* is already part of the environment, *string* replaces it; otherwise, the new *string* is added to the environment. A variable can be set to an empty value by specifying an empty *string*.

Do not free a pointer to an environment entry while the environment entry is still in use. The environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to **putenv**, then exit the function in which the variable is declared.

The **putenv** function returns 0 if it is successful. A return value of −1 indicates an error.

putenv

Example:

The following example tries to change an environment variable. If it fails, the example writes an error message.

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


if (putenv("PATH=a:\\bin;b:\\andy")== -1)
   {
   printf("putenv failed - out"
  "of environment space");
   exit(1);
   }
```

Related Topics:

getenv

Note: The **getenv** and **putenv** functions use the global variable *environ* to get access to the environment table. The **putenv** function can change the value of *environ*, thus invalidating the *envp* argument to the main function.

The environment manipulated by **putenv** is local to the process currently running. You cannot enter new items in your command level environment using **putenv**. When the program ends, the environment reverts to the parent process environment (DOS level in most cases). However, this environment is passed on to any child processes that are spawned, and they get any new items added using **putenv**.

Purpose:

Writes a given *string* to **stdout**.

Format:

#include <stdio.h>

int puts(*string*)
const char **string*; /* String to write to **stdout** */

Comments:

The **puts** function writes the given *string* to the standard output stream **stdout**, replacing the ending null character (\0) in the *string* with a newline character (\n) in the output stream.

The **puts** function returns the last character written, which is always the newline character. A return value of EOF indicates an error.

Example:
The following example writes a prompt to **stdout**:

```
#include <stdio.h>

int result;

result = puts("Insert data disk and press any key");
```

Related Topics:

fputs, **gets**

putw

Purpose:

Writes a binary value to the specified *stream*.

Format:

#include <stdio.h>

int putw(*binint*, *stream*)
int *binint*; /* Binary integer to write */
FILE **stream*; /* Pointer to file structure */

Comments:

The **putw** function writes a binary value of type **int** to the current position of the specified *stream*. The **putw** function does not affect the alignment of items in the *stream*, nor does it assume any special alignment.

The **putw** function returns the value written. A return value of EOF might indicate an error. Because EOF is also a legitimate integer value, use **ferror** to verify an error.

Example:

The following example writes a word to a data stream and checks for an error.

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

FILE *stream;
      .
      .
      .
putw(0345, stream);

if (ferror(stream)) {
   printf("putw failed");
   clearerr(stream);
   }
```

Related Topics:

getw

Note: The **putw** function is provided primarily for compatibility with previous libraries. Notice that portability problems might occur with **putw** because the size of an **int** and the ordering of bytes within an **int** differ across systems.

qsort

Purpose:

Uses a quick-sort algorithm to sort an array of elements.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

void qsort(base, num, width, compare)
void *base;
size_t num, width;
int (*compare)( const void *element1,const void *element2);
```

Comments:

The **qsort** function uses a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes in size. The *base* pointer is a pointer to the base of the array to be sorted. The **qsort** function overwrites this array with the sorted elements.

The *compare* pointer points to a routine, which you supply, that compares two array elements and returns an integer value specifying their relationship. The **qsort** function calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values.

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

The sorted array elements are stored in increasing order, as defined by your comparison routine. You can sort in reverse order by reversing the sense of "greater than" and "less than" in the comparison routine.

There is no return value.

Example:

This example reads the command-line parameters and uses **qsort** to sort them. It then displays the sorted arguments.

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

int qcompare(); /* function for qsort */

main(argc, argv)
int argc;
char **argv;
{
    char **result;
    int i;

/* Eliminate argv[0] from sort:    */
    argv++;
    argc--;

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

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

}

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

Related Topics:

bsearch

raise

Purpose:

Send a signal to a process.

Format:

#include <signal.h>

int raise (*sig*)
int *sig*; /*the constant expression for the signal */

Comments:

The raise function sends the signal *sig* to the running program. The return value is zero if successful, nonzero if unsuccessful.

Example:

```
#include <signal.h>
/*This program requests its own termination by raising condition
SIGTERM.*/
main ()
{
    raise(SIGTERM);         /*  Request termination.  */
}
```

Related Topics:

signal

Purpose:

Returns a pseudo random integer.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

int rand( )
```

Comments:

The **rand** function selects a pseudo random integer in the range 0 to 32767. Use the **srand** function before calling **rand** to set a random starting point. The default seed is 1.

The **rand** function returns a pseudo random number as described above. There is no error return.

Example:

```
#define  MAX       10
#include <math.h>
#include <stdlib.h>
main()
{
    int index[MAX];
    int i;

    for(i=0; i<MAX; i++)
  index[i]=i;

    shuffle(index,MAX);
    printf("The shuffled index vector was:");
    for(i=0; i<MAX; i++)
  printf(" %d",index[i]);
    printf("\n");
}
/* This subroutine takes the first
argument, a pointer to a vector of non-
negative integers, and performs an
in-place random permutation of the
elements of the vector.  The length is
the second argument.  The subroutine
uses the sign bits of the vector
elements as markers during the shuffle,
and reset them to zero (positive) after
the operation.  The constraints on the
numbers accepted after the random draw
are designed to make any of the n
factorial permutations equally probable.
*/

shuffle(m,n)
int m[];
int n;
{
    int i,j,k,r,cellcount,hold,save;
    unsigned multiplier;
    int cyclestart=-1;

    for (i=0; i<n; i++)
 /*  Test input vector for validity. */
  if(m[i]<0)
  {
```

```
  printf("Input error--element"
      " %d was negative.\n",i);
  return (-1);
  }
    cellcount=n;
a:  do
    {
  ++cyclestart;
  if(cyclestart>=n) goto wrapup;
    }
    while (m[cyclestart]<0);

    j=cyclestart;
    save=m[j];
b:  hold=save;
    frexp((float)cellcount,&k);
     /*   Which power of 2 is cellcount?  */
    multiplier=0x0001 << (15-k);
     /*   Form the multiplier.     */
    while((multiplier*cellcount-1)
  <= (r=rand()))  ;
    r %= cellcount;
    for (++r;  r>0; r--)
/*   The number of cells to skip.    */
 do (j==(n-1))?(j=0):(j++);
 while(m[j]<0) ;
/* Seek an uncanceled cell.    */
    save=m[j];
    m[j]=hold | 0x8000;
/*  Mark the cell with the sign bit. */
    cellcount--;
    if(j!=cyclestart) goto b;
/*  Is a cycle completed?             */
    goto a;

wrapup:
    for (i=0; i<n; i++)
/*  Reset all the sign bits.    */
  m[i] &= 0x7FFF;
    return (0);
}
```

Related Topics:

srand

read

Purpose:

Reads bytes from a file into a *buffer.*

Format:

```
  /* Required for function declarations */
#include <io.h>

int read(handle, buffer, count)
int handle;  /* Handle referring to open file */
char *buffer;  /* Storage location for data */
unsigned int count;  /* Maximum number of bytes */
```

Comments:

The **read** function tries to read *count* bytes from the file associated with *handle* into *buffer.* The read operation begins at the current position of any file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

The **read** function returns the number of bytes actually read, which can 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 incorrect. Or the file is not open for reading. Or the file is locked.

If you are reading more than 32K bytes from a file, the return value should be of the type **unsigned int**. However, the maximum number of bytes that can be read from a file is 65534, because 65535 (or 0xFFFF) is indistinguishable from -1, and returns an error.

If the file was opened in text mode, the return value might not correspond to the number of bytes actually read. When text mode is in effect, each carriage return/line feed pair is replaced with a single

line feed character. Only the single line feed character is counted in the return value. The replacement does not affect the file pointer.

Under OS/2 in large and compact models, memory is reserved from the OS/2 heap; each reserved unit is memory-protected and limited in size. If you give, in a **read** or **fread** call, a read count that is greater than the size of the allocated buffer, OS/2 issues a **General Protection Failure** message, even if the file being read is small enough to fit within the boundaries of the buffer.

Example:

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

char buffer[60000];

main()
{
   int fh, bytesread;
   unsigned int nbytes = 60000;

   if((fh = open("c:\\data\\conf.dat",

    O_RDONLY)) == -1) {
      perror("open failed on input file");
      exit(1);
      }
   if ((bytesread = read(fh,buffer,nbytes)) == -1)
      perror("");
   else
      printf("Read %u bytes from file\n",bytesread);
}
```

Related Topics:

creat, fread, open, write

Note: Under DOS, when files are open in text mode, a Ctrl + Z character is treated as an end-of-file indicator. When the Ctrl + Z is found, reading stops, and the next reading returns 0 bytes. You must close the file to clear the end-of-file indicator.

realloc

Purpose:

Changes the size of a previously-reserved block of storage.

Format:

```
 /* Required for function declarations */
#include <stdlib.h>

void *realloc(ptr, size)
 /* Pointer to previously-reserved storage block */
void *ptr;
size_t size;  /* New size in bytes */
```

Comments:

The **realloc** function changes the size of a previously-reserved storage block. The *ptr* 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 *ptr* argument can also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **malloc**, or **realloc** since the block was freed.

The **realloc** function returns a pointer to the reallocated storage block. The block might be moved when its size changed; thus, the *ptr* argument to **realloc** is not necessarily the same as the return value.

The return value is NULL if size is 0, or if there is not enough storage space to expand the block to the given size. The original block is freed when this occurs.

The storage space to which the return value points is guaranteed to be aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value.

Example:

The following example gets enough space for 50 characters. Then it reallocates the block to hold 100 characters.

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

char *alloc;

alloc = malloc(50*sizeof(char));
      .
      .
      .

if (alloc != NULL)
   alloc = realloc(alloc, 100*sizeof(char));
```

Related Topics:

calloc, free, malloc

remove

Purpose:

Deletes the specified file.

Format:

```
#include <stdio.h>

int remove(pathname)
  /* Path name of file to be removed */
const  char *pathname;
```

Comments:

The **remove** function deletes the file specified by *pathname*.

The **remove** function returns the value 0 if it successfully deletes the file. A return value of -1 indicates an error, and *errno* is set to one of the following values.

Value	Meaning
EACCESS	The pathname specifies a directory or a read-only file.
ENOENT	The file or pathname was not found, or (in OS/2 mode) the filename was improperly formed.

You may issue a remove call against an open file. The file is deleted when it is closed.

Example:

The following example removes the file TMPFILE and checks for an error, issuing a message if an error occurs.

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

int result;

result = remove("tmpfile");
if (result == -1)
   perror("could not delete tmpfile");
```

Related Topics:

close, unlink

rename

Purpose:

Renames a file or directory.

Format:

```
   /* Required for function declarations */
#include <stdio.h>

int rename(oldname,newname)
const char *newname; /* Pointer to new name */
const char *oldname; /* Pointer to old name */
```

Comments:

The **rename** function renames the file or directory specified by *oldname* to the name given by *newname*. The *oldname* pointer must specify the pathname of an existing file or directory. The *newname* pointer must not specify the name of an existing file or directory.

You can use the **rename** function to move a file from one directory to another by giving a different pathname in the *newname* argument. However, files cannot be moved from one device to another (for example, from drive A: to drive B:). Directories can only be renamed, not moved. The capability of renaming directories is available only under DOS 3.30 and OS/2.

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
EACCESS	The file or directory specified by *newname* already exists or could not be created (non-valid path). Or *oldname* is a directory and *newname* specifies a different path.
ENOENT	The filename or pathname specified by *oldname* not found, or, in OS/2 mode, the file was incorrectly named.
EXDEV	An attempt was made to move a file to a different device.

Example:

The following example changes name of the file INPUT to the name DATA:

```
#include <stdio.h>

int result;

result = rename("input", "data");
```

Related Topics:

creat, fopen, open

rewind

Purpose:

Repositions the file pointer to the beginning of a file.

Format:

#include <stdio.h>

void rewind(*stream*)
FILE **stream*; /* Pointer to file structure */

Comments:

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. A call to **rewind** is the same as:

(void) fseek(*stream*, 0L, SEEK_SET);

except that **rewind** clears the end-of-file and error indicators for the stream, but **fseek** does not clear the error indicators.

There is no return value.

Example:

This program first opens a file named DATA for input and output. It writes integers to the file. Next, it uses REWIND to reposition the file pointer to the beginning of the file, then reads the data back in.

```
#include <stdio.h>

FILE *stream;
int data1, data2;
main()
{
   data1 = 1; data2 = -37;

   /* Place data in the file */
stream = fopen("data", "w+");
fprintf(stream, "%d %d", data1, data2);

   /* Now read the data file */
rewind(stream);
fscanf(stream, "%d", &data1);
fscanf(stream, "%d",&data2);
printf("The values read back in are: %d and %d\n",
       data1, data2);
}
```

Related Topics:

fseek, ftell

rmdir

Purpose:

Deletes a directory.

Format:

```
 /* Required for function declarations */
#include <direct.h>
int rmdir(pathname)
  /* Path name of directory to remove  */
char *pathname;
```

Comments:

The **rmdir** function deletes the directory specified by *pathname*. The directory must be empty, and it must not be the current working directory or the root directory.

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
EACCESS	The given pathname is not a directory, the directory is not empty, or the directory is the current working directory or root directory.
ENOENT	The pathname was not found.

Example:

The following example deletes two directories: one at the root, and one in the current working directory:

```
#include <direct.h>

int result1, result2;

result1 = rmdir("\\data");
result2 = rmdir("data");
```

Related Topics:

chdir, mkdir

rmtmp

Purpose:

Cleans up the temporary files in the current directory.

Format:

#include <stdio.h>

int rmtmp( )

Comments:

The **rmtmp** function removes all those temporary files in the current directory that **tmpfile** creates. Use **rmtmp** only in the same directory in which the temporary files were created.

The **rmtmp** function returns the number of temporary files closed and deleted.

Example:

```
#include <stdio.h>

main()
   {
   int numdeleted;
   FILE *stream;
      .
      .
      .
   if ((stream = tmpfile( )) == NULL)
      perror("Could not open new temporary file");
      .
      .
      .
   numdeleted = rmtmp( );
   printf("Number of files closed and deleted"
   "in current directory = %d", numdeleted);
   }
```

Related Topics:

flushall, tmpfile, tmpnam

Purpose:

Resets the break value of the calling process.

Format:

```
 /* Required for function declarations */
#include <malloc.h>

void *sbrk(incr)
 /* Number of bytes added or subtracted */
int incr;
```

Comments:

The **sbrk** function resets the break value for the calling process. The *break value* is the address of the first byte of unallocated storage. The **sbrk** function adds *incr* bytes to the break value; the size of the storage reserved for the process is adjusted accordingly. The *incr* variable can be negative, in which case the amount of reserved space is decreased by *incr* bytes.

The **sbrk** function returns the old break value. A return value of (char *)-1 indicates an error and sets *errno* to ENOMEM, indicating there was not enough storage space.

Example:

The following example first adds an increment of 100 bytes to the break value of the process. Next, the call to **sbrk** reduces the reserved storage by 40 bytes, resulting in additional reserved storage of 60 bytes beyond the original break.

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

void *alloc;
alloc = sbrk(100);
      .
      .
      .
if (alloc != NULL)
   sbrk(-40);
```

sbrk

Related Topics:

calloc, free, malloc, realloc

Note: The address returned by **sbrk** is not necessarily aligned for storage of any particular type of object. In compact-, large-, and huge-model programs **sbrk** fails and returns (char *)-1. Use **malloc** for allocation requests in these models or when alignment is required.

Purpose:

Reads data from **stdin** into locations given by *arguments*.

Format:

```
#include <stdio.h>

int scanf(format-string [, argument...])
const char *format-string; /* Format-control string */
```

Comments:

The **scanf** function reads data from the standard input stream **stdin** into the locations given by *arguments*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields. The *format-string* can contain one or more of the following:

- White-space characters, which are (blank (' '), tab (\t), and new line (\n). A white-space character causes **scanf** to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in the *format-string* matches any combination of white-space characters in the input.
- Characters that are not white space, except for the percent sign character (%). A nonwhite-space character causes **scanf** to read, but not to store, a matching nonwhite-space character. If the next character in **stdin** does not match, **scanf** ends.
- 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 **scanf** function reads the *format-string* from left to right. Characters outside of format specifications are expected to match the sequence of characters in **stdin**; the matched characters in **stdin** are

scanned but not stored. If a character in **stdin** conflicts with the *format-string*, **scanf** ends. The conflicting character is left in **stdin** as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first *argument*. The second format specification converts the second input field and stores it in the second *argument*, and so on through the end of the *format-string*.

An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field *width* is reached, whichever comes first. If there are too many arguments for the given format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the given format specifications.

A format specification has the following form:

`%[*][width][F|N][h|l|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 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 a percent sign character, 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 are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional **F** and **N** prefixes cancel the default addressing conventions of the storage model that you are using. Prefix **F** to an *argument* pointing to a **far** object. Prefix **N** to an *argument* pointing to a **near** object. **F** and **N** are IBM extensions.

The optional prefix **l** shows that you use the **long** version of the following *type*, while the prefix **h** indicates that the short version is to be used. The corresponding *argument* should point to a **long** or **double** object (for the **l** character), a **long double** object (for the **L** character), or a **short** object (with the **h** character). The **l** and **h** modifiers may be used with the **d**, **i**, **n**, **o**, **x**, and **u** *type* characters. The **l** modifier may also be used with the **e**, **f** and **g** *type* characters. The **L** modifier may be used with the **e**, **f** and **g** *type* characters. The **l** and **h** modifiers are ignored if specified for any other *type*. The *type* characters and their meanings are in the following table.

Character	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to **int**
D	Decimal integer	Pointer to **long**
o	Octal integer	Pointer to **int**
O	Octal integer	Pointer to **long**
x	Hexadecimal integer	Pointer to **int**
X	Hexadecimal integer	Pointer to **long**
i	Decimal, hexadecimal, or octal integer	Pointer to **int**
I	Decimal, hexadecimal, or octal integer	Pointer to **long**
u	Unsigned decimal integer	Pointer to **unsigned int**
U	Unsigned decimal integer	Pointer to **unsigned long**
e,f,g,E,G	Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent ("e" or "E") followed by a possibly signed integer value	Pointer to **float**
c	Character; whitespace characters that are ordinarily skipped are read when **c** is specified; to read the next nonwhitespace 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 call to **scanf**
p	Value in the form *xxxx:yyyy*, where *x* and *y* are uppercase hexadecimal digits	Pointer to **far** pointer to void

To read strings not delimited by space characters, a set of characters in brackets ([ ]) can be substituted for the **s** (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification **%*n*c**, where *n* is a decimal integer. In this case, the **c** type character means 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 added.

The input for a **%x** format specifier is interpreted as a hexadecimal number; the leading 0x (0X) is not needed as it is in the language. Thus, if the input is 0xffff, the result returned from **scanf** is 0 because x is not a correct hex character. The correct input would be ffff. If the data file contains leading 0x (0X) characters, the correct format specifier is "0x%x".

The **scanf** function scans each input field character by character. It might stop reading a particular input field before it reaches a space character. The specified *width* has been reached, so the next character cannot be converted as specified, conflicts with a character in the control string that it should match, fails to appear, or appears in a given character set. When this occurs, the next input field begins at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on **stdin**.

The **scanf** function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

scanf

Examples:

The following example scans various types of data:

```
#include <stdio.h>

int i;
float fp;
char c, s[81];

scanf("%d %f %c %s", &i, &fp, &c, s);
```

The following example converts a hexadecimal or octal integer to a decimal integer. The **do** loop ends if the input value is 00 or if **scanf** is unable to assign the field.

```
#include <stdio.h>

main()
{
   int numassigned, val;

   printf("Enter a hexadecimal or octal"
         " number or 00 to quit:\n");
   do {
      printf("Number = ");
      numassigned = scanf("%i", &val);
      printf("Decimal number = %i\n", val);
      }
   while (val && numassigned);
}
```

Output:

```
Enter hexadecimal or octal number or 00 to quit:
Number = 0xf
Decimal number = 15
Number = 0100
Decimal number = 64
Number = 00
Decimal number = 0
```

Related Topics:

fscanf, printf, sscanf

Purpose:

Fills a structure with the current contents of segment registers. Under OS/2, references to segments are translated to selector values.

Format:

#include <dos.h>
void segread(*segregs*)
 /* Segment register values */
struct SREGS **segregs*;

Comments:

The **segread** function fills the structure to which *segregs* points with the current contents of the segment registers. Use **segread** under DOS with **intdosx** nd **int86x** functions to retrieve segment register values for later use.

There is no return value.

Example:

This program reads the four segment registers cs, ds, es, and ss. It prints their values as hexadecimal numbers.

```
#include <dos.h>

struct SREGS segregs;
unsigned int cs, ds, es, ss;

main()
{
    segread(&segregs);
    cs = segregs.cs;
    ds = segregs.ds;
    es = segregs.es;
    ss = segregs.ss;
    printf("In hexadecimal the four"
           " segment values are:  %x"
           " %x  %x  %x\n",
           cs,ds,es,ss);
 }
```

segread

Related Topics:

intdosx, int86x, FP_SEG

Purpose:

Allows control of buffering.

Format:

#include <stdio.h>
void setbuf(*stream, buffer*)
FILE **stream*; /* Pointer to file structure */
char **buffer*; /* User-allocated buffer */

Comments:

The **setbuf** function lets you control buffering for the specified *stream*. The *stream* pointer must refer to an open file. If the *buffer* argument is NULL, the *stream* is unbuffered. If not, the *buffer* must point to a character array of length BUFSIZ, which is the buffer size defined in **stdio.h**. The system uses the *buffer*, which you specify, for input/output buffering instead of the default system-allocated buffer for the given *stream*.

The **stderr** and **stdaux** streams are unbuffered by default but can be assigned buffers with **setbuf**.

There is no return value.

setbuf

Example:

The following example opens the file DATA1 for reading and DATA2 for writing. It then calls the **setbuf** function for each data stream, establishing a buffer of length BUFSIZ for the first file and no buffering for the output file. For **stdio.h**, the value of the manifest constant BUFSIZ is 512.

```
#include <stdio.h>

char buf[BUFSIZ];
FILE *stream1, *stream2;
stream1 = fopen("data1", "r");
stream2 = fopen("data2", "w");
setbuf(stream1,buf);
setbuf(stream2,NULL);
```

Related Topics:

fflush, fopen, fclose

Purpose:

Saves a stack environment that can be restored by **longjmp**.

Format:

```
#include <setjmp.h>

int setjmp(env)
   /* Variable in which environment is stored */
jmp_buf env;
```

Comments:

The **setjmp** function saves a stack environment that can subsequently be restored by **longjmp**. The **setjmp** and **longjmp** functions provide a way to perform a nonlocal **goto**. This use of a nonlocal **goto** passes control to error-handling or recovery code in a previously called function without using the normal calling or return conventions.

A call to **setjmp** causes it to save the current stack environment in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable.

The **setjmp** function returns the value 0 after saving the stack environment. If **setjmp** returns as a result of a **longjmp** call, it returns the *value* argument of **longjmp**, or 0 if the *value* argument of **longjmp** is 1. There is no error-return value.

Example:

The following example provides for saving the stack environment at the statement:

if(setjmp(mark) != 0) ...

When the system first performs the **if** statement, it saves the environment in *mark* and sets the condition to FALSE because **setjmp** returns a 0 when it saves the environment. The system prints the message:

setjmp has been called

The subsequent call to function **p** tests for a local error condition, which can cause it to perform the **longjmp** function. Then, control returns to the original **setjmp** function using the environment saved in *mark*. This time the condition is TRUE because -1 is the return value from the **longjmp** function. The example then performs the statements in the block and prints:

longjmp has been called

Then, it performs your **recover** function and leaves the program.

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

jmp_buf mark;

main()
{
   if (setjmp(mark) != 0) {
      printf("longjmp has been called\n");
      recover();
      exit(1);
      }
   printf("setjmp has been called\n");
         .
         .
         .
   p();
         .
         .
         .
}
p()
{
   int error = 0;
         .
         .
         .
   if (error != 0)
      longjmp(mark, -1);
         .
         .
         .
}
recover()
{
    /* Put code here that ensures that *
     * exiting the program does not    *
     * corrupt the data files.     */
         .
         .
         .
}
```

Related Topics:

longjmp

CAUTION:
The values of register variables in the function calling setjmp might not be restored to the proper values after a longjmp call is run.

setmode

Purpose:

Sets translation mode of a file.

Format:

```
#include <fcntl.h>
 /* Required for function declarations */
#include <io.h>

int setmode(handle, mode)
int handle;  /* File handle */
int mode;  /* New translation mode */
```

Comments:

The **setmode** function sets the translation mode of the file given by *handle* to *mode*. The *mode* must be one of the manifest constants in the following table:

Manifest Constant	Meaning
O_TEXT	Set text (translated) mode. Carriage-return/line-feed combinations are translated into a single line feed on input. Line-feed characters are translated into carriage-return/line-feed combinations on output.
O_BINARY	Set binary (untranslated) mode. The above translations are suppressed.

Use the **setmode** function to change the default translation mode of **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, or you can use it on on any file.

If successful, **setmode** returns the previous translation mode. A return value of SIG_EER indicates an error, and *errno* is set to one of the following values.

Value	Meaning
EBADF	Non-valid file handle
EINVAL	Non-valid *mode* argument (neither O_TEXT nor O_BINARY)

Example:

The following example sets the translation mode of the standard input file **stdin** to binary (untranslated mode).

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

int result;

result = setmode(fileno(stdin), O_BINARY);
```

Related Topics:

creat, fopen, open

setvbuf

Purpose:

Controls buffering and buffer size for a specified stream.

Format:

```
#include <stdio.h>

int setvbuf(stream,buf,type,size)
FILE *stream;  /* Pointer to file structure */
char *buf;     /* User-allocated buffer */
int type; /* Type of buffer:    *
                    * _IONBF for no buffer     *
                    * _IOFBF for full buffering *
                    * _IOLBF for line buffering */
size_t size;   /* Size of buffer */
```

Comments:

The **setvbuf** function controls both buffering and buffer size for the specified *stream*. The *stream* must refer to an open file. The array to which *buf* points is used as the buffer, unless it is NULL, in which case the stream is unbuffered. If the stream is buffered, the *type* specified is used; the type must be _IONBF, _IOFBF, or _IOLBF. If the type is _IOFBF or _IOLBF, then *size* is the size of the buffer. If type is _IONBF, the stream is unbuffered, and *size* and *buf* are ignored.

Value	Meaning
_IONBF	No buffer is used regardless of *buf* or *size*.
_IOFBF	Full buffering is used unless *buf* is NULL. Use *buf* as the buffer and *size* as the size of the buffer.
_IOLBF	Same buffering as _IOFBF.

The legal values for *size* are greater than 0 and less than the maximum integer size.

The **setvbuf** function returns a 0 if successful and nonzero if you specify an illegal type or buffer size.

Example:

```
#include <stdio.h>

char buf[1024];
FILE *stream1, *stream2;

main()
   {
   stream1 = fopen("data1", "r");
   stream2 = fopen("data2", "r");
/* Stream1 uses a user-assigned buffer of  *
 * 1024 bytes, while stream2 is unbuffered */

   if (setvbuf(stream1, buf, _IOFBF, sizeof(buf))
       !=0)
      printf("Incorrect type or size of buffer1\n");
   if (setvbuf(stream2, NULL, _IONBF, 0) !=0)
      printf("Incorrect type or size of buffer2\n");
   }
```

Related Topics:

setbuf, fflush, fopen, fclose

signal

Purpose:

Allows handling of an interrupt signal from operating system.

Format:

```
#include <signal.h>

void (*signal(sig, func))(sig)
void (*func)(sig);    /* Function to run  */
int sig;    /* Signal value */
```

Comments:

The **signal** function allows a process to choose one of several ways to handle an interrupt signal from the operating system. Under DOS, the *sig* argument must be one of the manifest constants SIGINT or SIGFPE, defined in **signal.h**. SIGINT corresponds to the DOS interrupt signal, INT 23H (the Control + C signal). SIGFPE corresponds to floating-point exceptions that are not masked, such as overflow, division by zero, or non-valid operations. The *func* argument must be one of the manifest constants, defined in **signal.h**, SIG_DFL, SIG_IGN, or a function address.

Under OS/2, the *sig* argument must be one of the manifest constants, SIGABRT, SIGILL, SIGSEGV, SIGINT, SIGFPE, SIGTERM, SIGUSR1, SIGUSR2, SIGUSR3, or SIGBREAK, defined in **signal.h**. The *func* argument must be one of the manifest constants, SIG_DFL, SIG_IGN, SIG_ERR, or SIG_ACK, defined in **signal.h**, or a function address.

The meaning of the values of *sig* are as follows:

Value	Meaning
SIGABRT	SIGABRT is the abnormal termination signal sent by the **abort( )** function. Default action is to end the program.
SIGBREAK	SIGBREAK is a Control + Break signal that you can use only on OS/2. Default action is to end the program.

SIGFPE SIGFPE corresponds to floating-point exceptions that are not masked, such as overflow, division by zero, and invalid operation. Default action is to end the program.

SIGILL SIGILL corresponds to detection of an invalid function image. Under DOS, the SIGILL signal handler is reset to the default when a user-defined handling routine is called. Under OS/2, the SIGILL signal handler is NOT reset to the default when a user-defined handling routine is called.

SIGINT SIGINT is a Control + C signal. In DOS, SIGINT corresponds to the DOS INT 23H interrupt signal, and in OS/2, SIGINT corresponds to the OS/2 SIGINTR signal. Default action is to end the program.

SIGSEGV SIGSEGV corresponds to an invalid access to memory. Default action is to end the program.

SIGTERM SIGTERM corresponds to the OS/2 SIGTERM program termination signal, which can be sent only by using the OS/2 DOSKILLPROCESS command. Default action is to end the program.

SIGUSR1 SIGUSR1 corresponds to the OS/2 process-flag A signal, which you can use only on OS/2. Default action is to ignore the signal.

SIGUSR2 SIGUSR2 corresponds to the OS/2 process-flag B signal, which you can use only on OS/2. Default action is to ignore the signal.

SIGUSR3 SIGUSR3 corresponds to the OS/2 process-flag C signal, which you can use only on OS/2. Default action is to ignore the signal.

SIGUSR1, SIGUSR2, and SIGUSR3 are signals that you can define and send using DOSFLAGPROCESS. For more information about using these signal, see the DOSCALLS section in *IBM Operating System/2 Technical Reference.*

The action taken when the interrupt signal is received depends on the value of *func.*

Value	Meaning
SIG_ACK	SIG_ACK acknowledges receipt of a signal. A process uses SIG_ACK to re-enable recognition of a given signal whenever a signal handler is prepared to receive more signals of this type. This option does not affect the handler installed for a given signal. You can use SIG_ACK only on OS/2. In a process with multiple threads of performance, only the first thread receives a signal, though the action taken is to call the function to which any of the threads of that process last set the **func** argument.
SIG_DFL	On DOS, the calling process is ended and control returns to the DOS command level. All files opened by the process are closed, but buffers are not flushed. On OS/2, the system default response is to ignore all signals except SIGTERM, SIGBREAK, SIGSEGV, SIGINT, SIGABRT, and SIGFPE. The system responses for SIGTERM and SIGFPE are the same as those for DOS.
SIG_SGE	SIG_SGE is equivalent to SIG_IGN except that it also makes it an error for a process to send the signal to this process using DOSFLAGPROCESS, if the signal is SIGUSR1, SIGUSR2, or SIGUSR3, or using DOSKILLPROCESS if the signal is SIGTERM. It has exactly the same effect as SIG_IGN for SIGINT and SIGBREAK signals because these can only be sent by the operating system. You can use SIG_SGE only on OS/2.
SIG_IGN	The interrupt signal is ignored. Never give this value with SIGFPE because the floating-point state of process is left undefined.
Function address	The function address installs a specified function as a handler for the given signal. For SIGINT signals on DOS, the function pointed to by *func* is passed the single argument SIGINT and run. If the function returns, the calling process resumes running just after the point where it received the interrupt signal. Before the specified function is

run, 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 you to reset signals in the called function if desired.

For all signals other than SIGFPE on OS/2, the function is passed two arguments, the signal number and, if appropriate, the argument furnished by the DOSFLAGSPROCESS. The second argument is appropriate only if the signal is SIGUSR1, SIGUSR2, or SIGUSR3.

For SIGFPE signals on all versions of DOS, the function to which *func* points is passed the single argument, SIGFPE, then performed. (See the include file **float.h** for definitions of the FPE_*xxx* subcodes.) The value of *func* is not reset on receiving the signal; to recover from floating-point exceptions, use **setjmp** in connection with **longjmp**. (See the example under **_fpreset** in this chapter.) If the function returns, the calling process resumes running with the floating-point state of the process left in an undefined state.

For all types of signals and for all versions of OS/2, if the function returns, the calling process resumes running immediately following the point at which it received the interrupt signal.

On DOS before the specified function runs, the operating system sets the value of *func* to SIG_DFL. It then treats the next interrupt signal as SIG_DFL unless an intervening call to SIGNAL specifies otherwise. This action lets you reset signals in the called function, if necessary.

OS/2 does not reset the signal handler to the system default response. Instead, OS/2 ensures that a process can receive no signals of a given type until the process sends a SIG_ACK to the operating system. You can restore the default system response from the handler by sending a

SIG_DFL and then a SIG_ACK to the operating system.

On DOS, the **signal** function returns the previous value of *func*.

On OS/2, for SIGFPE signals, the **signal** function returns the previous value of *func* for SIGFPE. For all other signals, the return value is as follows:

Previous func Value	**Current func Value**
SIG_IGN	SIG_IGN
SIG_DFL	SIG_DFL
SIG_SGE	SIG_SGE
SIG_ACK	The address of the currently installed handler
Function address	The function address.

A return value of SIG_ERR indicates an error, and *errno* is set to EINVAL, indicating an incorrect *sig* value.

The possible causes of error are an incorrect *sig* value, a value for *func* that is less than SIG_ACK but undefined, or a value for *func* of SIG_ACK when no handler is currently installed.

Example:

In the following example, the call to **signal** in **main( )** establishes the routine **handler( )** to process the DOS interrupt signal when it occurs. An error value returned from this call to **signal( )** causes the program to end with a printed error message. The **handler( )** routine asks you to enter a **Y** or **y** from the keyboard if you want to halt the program. Entering any other character causes the program to resume operation.

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

int handler();

main()
{
   if (signal(SIGINT,handler) == (int(*)())-1) {
      perror("Could not set SIGINT");
      abort();
      }
            .
            .
            .
}

int handler()
{
   char ch;

   signal(SIGINT,handler);
   printf("End processing?");
   ch = getchar ( );
   if (ch == 'y' || ch == 'Y')
      exit(0);
}
```

Related Topics:

abort, exit, _exit, _fpreset, raise, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

Note: Signal settings are not preserved in child processes created by calls to **exec** or **spawn** functions. The signal settings are reset to the default in the child process. Calling library functions from within a signal handler routine may result in undefined behavior.

sin - sinh

Purpose:

Return the sine and hyperbolic sine.

Format:

```
#include <math.h>

double sin(x)    /* Calculate sine of x */
 /* Calculate hyperbolic sine of x */
double sinh(x)
double x;  /* Angle in radians */
```

Comments:

The **sin** and **sinh** functions return the sine and hyperbolic sine of *x*, respectively.

The **sin** function returns the sine of *x*. If *x* is large, a partial loss of significance in the result might occur. In such cases, **sin** generates a PLOSS error, but no message is printed. If *x* is so large that a total loss of significance results, **sin** prints a TLOSS error 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 the value HUGE_VAL (positive or negative, depending on the value of *x*).

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

Example:

The following example computes *y* as the sine of $\pi/2$ and *z* as the hyperbolic sine of $\pi/2$:

```
#include <math.h>

double pi, x, y, z;

pi = 3.1415926535;
x = pi/2;
y = sin(x);     /* y is 1.0 */

⋮
z = sinh(x);    /* z is 2.3 */
```

Related Topics:

acos, asin, atan, atan2, cos, cosh, tan, tanh

sopen

Purpose:

Opens a file for shared reading or writing.

Format:

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>
  /* Required for function declarations */

int sopen(pathname, oflag, shflag [, pmode])
char *pathname;  /* File path name */
int oflag;  /* Type of operations allowed */
int shflag;  /* Type of sharing allowed */
int pmode;  /* Permission setting */
```

Comments:

The **sopen** function opens the file specified by *pathname* and prepares the file for subsequent shared reading or writing as defined by *oflag* and *shflag*. The *oflag* is an integer expression formed by combining one or more of the manifest constants defined in **fcntl.h**. When more than one manifest constant is given, the constants are joined with the OR operator (|).

Note: File sharing modes do not work correctly for buffered files, so do not use **fdopen** to associate a file opened for sharing (or locking) with a stream.

The *oflag* of every **sopen** must contain one of the following:

O_RDONLY
O_WRONLY
O_RDWR

The six other manifest constants modify how the file is processed.

Oflag	Meaning
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_BINARY	Open the file in binary (untranslated) mode. (See **fopen** for a description of binary mode.)
O_CREAT	Create and open a new file. This has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.
O_RDWR	Open the file for both reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TEXT	Open the file in text (translated) mode. (See **fopen** for a description of text mode.)
O_TRUNC	Open and truncate an existing file to 0 length. The file must have write permission, and the contents of the file are destroyed.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.

CAUTION:
O_TRUNC destroys the complete contents of an existing file. Use with care.

The *shflag* function is a constant expression consisting of one of the following manifest constants, defined in **share.h**. If SHARE.COM (or SHARE.EXE under OS/2) has not been installed, the system ignores the sharing mode. See your DOS documentation for detailed information on sharing modes.

Shflag	Meaning
SH_COMPAT	Set compatibility mode (not available under OS/2)
SH_DENYRW	Deny read and write access to file
SH_DENYWR	Deny write access to file
SH_DENYRD	Deny read access to file
SH_DENYNO	Permit read and write access (OS/2 only).

Under DOS, the default value of *shflag* is SH_COMPAT, while under OS/2 it is SH_DENYNO. The *pmode* argument is required only when O_CREAT is specified. If the file does not exist, *pmode* specifies the permission settings of the file, which are set when the new file is closed for the first time. Otherwise, the *pmode* argument is ignored. The *pmode* argument is an integer expression containing one or both of the manifest constants S_IWRITE and S_IREAD. When both constants are given, they are joined with the OR operator |. The meanings of the *pmode* argument are in the following table.

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 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.

Specifying a *pmode* of S_IREAD is similar to making a file read-only with the DOS ATTRIB command.

The **sopen** function applies the current file permission mask to *pmode* before setting the permissions (see **umask** in this chapter).

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
EACCESS	The given pathname is a directory, but the file is read-only and an open for writing was attempted, or a sharing violation occurred.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available. (There are too many open files.)
ENOENT	File or pathname was not found.

Note: When opening text files, the compiler opens the file for read/write access to look for a Ctrl+Z at the end. Because of this, an open will fail if the file has previously been opened with a DENY_RD sharing mode, even if the open specified write only access.

sopen

Examples:

This program first checks the version of DOS in use. If it is 3.00 or later, the programmer uses **sopen** to open a file called DATA for sharing.

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

extern unsigned char _osmajor;
int fh;
main()
{

/*    The _osmajor variable is used to test     *
 * the DOS version number before calling sopen. */
if (_osmajor >= 3)
  /* Open for sharing. */
   fh = sopen("data", O_RDWR | O_BINARY, SH_DENYR  W);
else
  /* No sharing */
   fh = open("data", O_RDWR | O_BINARY);
}
```

Related Topics:

close, creat, fopen, open, umask

Purpose:

Starts and runs a new child process.

Format:

```
#include <process.h>

int spawnl(modeflag, pathname,arg0,
        arg1,..., argn, NULL)
int spawnle(modeflag, pathname, arg0,
        arg1,..., argn, NULL, envp)
int spawnlp(modeflag, pathname, arg0,
         arg1,..., argn, NULL)
int spawnlpe(modeflag, pathname, arg0,
          arg1,...,argn, NULL, envp)

int spawnv(modeflag, pathname, argv)
int spawnve(modeflag, pathname, argv, envp)
int spawnvp(modeflag, pathname, argv)
int spawnvpe(modeflag, pathname, argv, envp)

  /* Execution mode for parent process  */
int modeflag;
  /* Path name of file to be executed */
char *pathname;
  /* List of pointers to arguments */
char *arg0,*arg1,..., *argn;
  /* Array of pointers to arguments */
char *argv[ ];
  /* Array of pointer to environment settings */
char *envp[ ];
```

Comments:

The **spawn** functions create and run a new child process. There must be enough storage available for loading and running the child process. The *modeflag* argument determines the action taken by the parent process before and during the **spawn**. The following values for *modeflag* are defined in **process.h**.

Value	Meaning
P_WAIT	Suspend the parent process until the performance of the child process is complete.
P_NOWAIT	Continue to run the parent process concurrently with the child process (OS/2 only).
P_OVERLAY	Overlay the parent process with the child process, erasing the parent process. (This is the same effect as **exec** calls.)

The *pathname* argument specifies the file run as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *pathname* does not have a filename extension or end with a period (.), the **spawn** routines first add the extension .COM and search for the file; if the search is unsuccessful, the extension .EXE is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the **spawn** routines search for *pathname* with no extension. The **spawnlp**, **spawnlpe**, **spawnvp**, and **spawnvpe** functions search for *pathname* (using the same procedures) in the directories specified by the PATH environment variable.

Pass arguments to the child process by giving one or more pointers to character strings as arguments in the **spawn** routine. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes. The ending 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 can be passed as separate arguments (**spawnl, spawnle**, **spawnlp**, and **spawnlpe**) or as an array of pointers (**spawnv**, **spawnve**, **spawnvp**, and **spawnvpe**). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. However, a different value will not produce an error. The *pathname* is available as *arg0* or *argv*[0] under DOS and OS/2.

Use the **spawnl**, **spawnle**, **spawnlp**, and **spawnlpe** routines in cases where you know the number of arguments. The *arg0* is usually a pointer to *pathname*. The *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** functions are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. The *argv*[0] is usually a pointer to the *pathname*. The *argv*[1] through *argv*[*n*] are pointers to the character strings forming the new argument list. The *argv*[*n*+1] must be a NULL pointer to mark the end of the argument list.

Files that are open when a **spawn** call is made remain open in the child process. In the **spawnl**, **spawnlp**, **spawnv**, **spawnvp** calls, and the child process inherits the environment of the parent. The **spawnle**, **spawnlpe**, **spawnve**, and **spawnvpe** functions let you alter the environment for the child process by passing a list of environment settings through the *envp* argument. *Envp* is an array of character pointers, each element of which points to a string, ended by a null, that defines an environment variable. Such a string has the form

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotes.) 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.

Under DOS, the return value is the exit status of the child process. The exit status is 0 if the process ended normally. The exit status can also be set to a nonzero value if the child process specifically calls

the **exit** function with a nonzero argument. If not set, a positive exit status indicates an abnormal exit with an **abort** or an interrupt. Under OS/2 the return from a spawn has one of two different meanings. The return value of a synchronous spawn is the exit status of the child process. The return value of an asynchronous spawn is the process identification of the child process. You can use the **wait** or **cwait** functions to get the child process exit code.

A return value of -1 indicates an error (the child process is not started), and *errno* is set to one of the following values.

Value	Meaning
E2BIG	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K bytes.
EINVAL	The *modeflag* argument is incorrect.
ENOENT	The file or path name was not found, or in OS/2 mode was incorrectly specified.
ENOEXEC	The specified file is not executable or has an incorrect executable file format.
ENOMEM	Not enough storage is available to run the child process.

Note: The spawn functions pass on all information on open files to the child process, including the translation mode. This is done using the environment passed to the child, which has a special entry called ";C_FILE_INFO". It is normally processed and deleted from the environment by the C startup code. However, if **spawn***xx* is used to create a non-C process (such as the command interpreter) and the environment strings are examined, this entry is still present. Also, because the information is passed in binary form, printing the environment shows some graphics characters in the definition string for this entry.

Signal settings are not preserved in child processes created by calls to **spawn** functions. The signal settings are reset to the default in the child process.

Example:

This example shows calls to four of the eight spawn routines. When called without arguments from the command line, the program first runs the code for case PARENT. It spawns a copy of itself, waits for its child to run, then spawns a second child. The instructions for the child are blocked to run only if *argv*[0] and one parameter were passed (case CHILD). In its turn, each child spawns a grandchild as a copy of the same program. The grandchild instructions are blocked by the existence of two passed parameters. The grandchild is permitted to overlay the child. Each of the processes prints a message identifying itself. This program runs under either DOS or OS/2; it does not use the P_NOWAIT mode flag setting for concurrent execution.

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

#define   PARENT        1
#define   CHILD         2
#define   GRANDCHILD    3

extern char **environ;

main(argc, argv, envp)
int  argc;
char **argv;
char **envp;
{
    int    result;
    char   *args[4];

    switch(argc)
    {
    case PARENT:
        /* no argument was passed: */
        /* spawn child and wait    */
        result = spawnle(P_WAIT, argv[0],
                argv[0], "one", NULL, envp);
        if (result)
            abort();
        /* spawn another child,    */
        /* and wait for it         */
        args[0] = argv[0];
        args[1] = "two";
        args[2] = NULL;
        result = spawnve(P_WAIT, argv[0],
                args, environ);
        if (result)
            abort();
        printf("Parent process ended\n");
        exit(0);
```

```
    case CHILD:
        /* one argument was passed: */
        /* allow grandchild to      */
        /* overlay                  */
        printf("child process %s began\n",
            argv[1]);
        /* child one ?              */
        if (*argv[1] == 'o')
        {
            spawnl(P_OVERLAY, argv[0],
                argv[0], "one", "two",
                NULL);
            abort();
        /* not executed because     */
        /* child was overlaid       */
        }
        /* child two ?              */
        if (*argv[1] == 't')
        {
            args[0] = argv[0];
            args[1] = "two";
            args[2] = "one";
            args[3] = NULL;
            spawnv(P_OVERLAY, argv[0],
                args);
            abort();
        /* not executed because     */
        /* child was overlaid       */
        }
        /* argument not valid       */
        abort();

    case GRANDCHILD:
        /* two arguments            */
        printf("grandchild %s ran\n",
            argv[1]);
        exit(0);
    }
}
```

Related Topics:

abort, cwait, execl, execle, execlp, execv, execve, execvp, exit, _exit, wait

Purpose:

Formats and stores characters or values in *buffer*.

Format:

```
#include <stdio.h>
int sprintf(buffer,format-string[, argument...])
char *buffer;    /* Storage location for output */
const char *format-string;    /* Format control string *
```

Comments:

The **sprintf** function formats and stores a series of characters and values in *buffer*. Any *argument* is converted and put out according to the corresponding format specification in the *format-string*. The *format-string* consists of ordinary characters and has the same form and function as the *format-string* argument for the **printf** function. See the **printf** keyword page for a description of the *format-string* and arguments.

The **sprintf** function returns the number of characters printed, not counting the ending null.

sprintf

Example:

```
#include <stdio.h>
char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;
main()
{
   c = 'l';
   i = 35; fp = 1.7320508;

/* Format and print various data  */
j = sprintf(buffer, "%s\n", s);
j += sprintf(buffer+j, "%c\n", c);
j += sprintf(buffer+j, "%d\n", i);
j += sprintf(buffer+j, "%f\n", fp);
printf("string:\n%s\ncharacter count = %d\n",
       buffer, j);
}
```

Related Topics:

fprintf, printf, sscanf

Purpose:

Calculates the square root.

Format:

#include <math.h>

double sqrt(*x*)
double *x*; /* Non-negative floating-point value */

Comments:

The **sqrt** function calculates the non-negative square root of *x*.

The **sqrt** function returns the square root result. If *x* is negative, the function prints a DOMAIN error message to **stderr**, sets *errno* to EDOM, and returns 0.

You can change error handling by using the **matherr** routine.

Example:

The following example computes *z* as the square root of the quantity *x*+*y*, printing an error message if *x*+*y* is negative.

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

double x, y, z;
      .
      .
      .
if ((z = sqrt(x+y)) == 0.0)
   if ((x+y) < 0.0)
      perror("sqrt of a negative number");
```

Related Topics:

exp, log, matherr, pow

srand

Purpose:

Sets the starting point for pseudo-random integers.

Format:

```
 /* Required for function declarations */
#include <stdlib.h>
void srand(seed)
   /* Seed for random number generation */
unsigned int seed;
```

Comments:

The **srand** function sets the starting point for producing a series of pseudo-random 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 retrieves the pseudo-random numbers generated.

There is no return value.

Example:

First, this program calls **srand** with a value other than 1 to initiate the random value sequence. Then the program computes 20 random values for the array of integers called **ranvals**.

```
#include <stdlib.h>
#include <stdio.h>
main()
{
int x, ranvals[20];

/* Initialize the random number generator *
 *      and save the first 20 random      *
 *     numbers generated in an array.      */

srand(17);
for (x = 0; x < 20; ranvals[x++] = rand())
printf("Iteration %d ranvals [%d] = %d\n",
       x+1, x, ranvals[x]);
}
```

Related Topics:

rand

sscanf

Purpose:

Reads data from a buffer into locations given by arguments.

Format:

#include <stdio.h>

int sscanf(*buffer, format-string* [,*argument*...])
const char **buffer*; /* Stored data */
const char **format-string*; /* Format control string *

Comments:

The **sscanf** function reads data from *buffer* into the locations given by *argument*s. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function. See the **scanf** reference page for a description of the *format-string*.

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.

Example:

This program uses **sscanf** to read various data from a string named *tokenstring*, then displays it.

```
#include <stdio.h>

char *tokenstring = "15 12 14...";
int i;
float fp;
char s[81];
char c;
main()
{

/* Input various data. */
sscanf(tokenstring, "%s", s);
sscanf(tokenstring, "%c", &c);
sscanf(tokenstring, "%d", &i);
sscanf(tokenstring, "%f", &fp);

/* Display the data */
printf("string = %s\n",s);
printf("character = %c\n",c);
printf("integer = %d\n",i);
printf("floating-point number = %f\n",fp);
}
```

Related Topics:

fscanf, scanf, sprintf

stackavail

Purpose:

Reports available stack space.

Format:

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

size_t stackavail ( )
```

Comments:

The **stackavail** function returns the approximate size in bytes of the stack space available for dynamic storage allocation with **alloca**.

The **stackavail** function returns the size in bytes as an unsigned integer value.

Example:

```
#include <malloc.h>


main()
   {
   char  *ptr;
   printf("Stack memory available before
    " alloca =  %u\n", stackavail( ));
   ptr = alloca (1000*sizeof(char));
   printf ("Stack memory available after"
        " alloca =  %u\n", stackavail());
   }
```

Output:

(Actual numbers may vary slightly.)

```
Stack memory available before alloca = 1682
Stack memory available after alloca = 678
```

Related Topics:

alloca, **freect**, **memavl**

Purpose:

Obtains information about a file or directory.

Format:

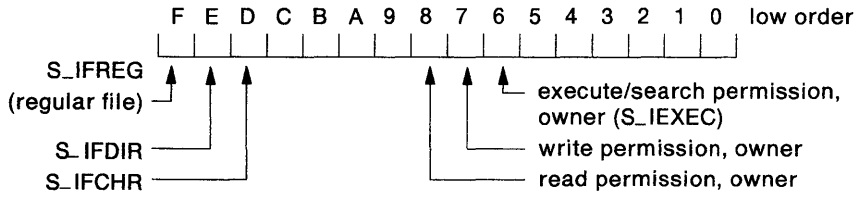
```
#include <sys\types.h>
#include <sys\stat.h>

int stat(pathname, buffer)
  /* Path name of existing file */
char *pathname;
  /* Pointer to structure to receive results */
struct stat *buffer;
```

Comments:

The **stat** function obtains information about the file or directory specified by *pathname* and stores it in the structure to which *buffer* points. The **stat** structure, defined in **sys\stat.h,** contains the following fields.

Field	Value
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 permission mode of the file. User run bits are set using the filename extension.



st_dev Drive number of the disk containing the file.

st_rdev Drive number of the disk containing the file (same as **st_dev**).

st_nlink Always 1.

st_size Size of the file in bytes.

st_atime Time of last modification of file.

st_mtime Time of last modification of file (same as **st_atime**).

st_ctime Time of last modification of file (same as **st_atime** and **st_mtime**).

There are three additional fields in the **stat** structure type that do not contain meaningful values under DOS.

The **stat** function returns the value 0 if the file status information is obtained. A return value of -1 indicates an error, and *errno* is set to ENOENT, indicating that the filename or pathname could not be found. ENOENT might also be set in OS/2 MODE when the filename was specified with more than 8 characters, or the extension with more than 3 characters.

Example:

The following example requests that the status information for the file CHILD.EXE be placed into the structure *buf*. If the request is successful and the file is executable, the example runs CHILD.EXE.

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

struct stat buf;
int result;
char *args[4];
       .
       .
       .
result = stat("child.exe", &buf);

if (result == 0)
   if (buf.st_mode & S_IEXEC)
      execv("child.exe", args);
```

Related Topics:

access, fstat

Note: If the given *pathname* specifies only a device, (for example "C:"), **stat** returns an error; fields in the **stat** structure are not meaningful.

_status87

Purpose:

Gets the floating-point status word.

Format:

```
#include <float.h>

 /* Get floating-point status word */
unsigned int _status87( )
```

Comments:

The **_status87** function gets the floating-point status word. The floating-point status word is a combination of the Numeric Coprocessor status word and other conditions detected by the numeric exception handler, such as floating-point stack underflow and overflow.

The bits in the value returned show the floating-point status. See the discussion of the **float.h** include file for a complete definition of the bits returned by _STATUS87.

Example:

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

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

main()
   {
   printf("status = %.4x - clear\n",_status87( ));
   y = a;
     /* Store into y is inexact, so underflow */
   printf("status = %.4x - inexact, underflow\n",
          _status87( ));
   b = y;   /* y is denormal */
   printf("status = %.4x - inexact, underflow,
          denormal\n", _status87());
   _clear87(); /* Clear user 8087 status */
   }
```

Related Topics:

_clear87, _control87

Purpose:

Operates on null-ended strings. /stricmp /strcat

Format:

```
  /* Required for function declarations */
#include <string.h>

  /* Append string2 to string1 */
char *strcat(string1, string2)
char *string1;  /* Destination string */
const char *string2;  /* Source string */

  /* Search for first occurrence of *
   *        c in string        */
char *strchr(string, c)
const char *string;  /* Source string */
int c;  /* Character to be located */

  /* Compare strings                  */
int strcmp(string1, string2)
const char *string1;
const char *string2;

  /* Copy string2 to string1 */
char *strcpy(string1, string2)
char *string1;  /* Destination string */
const char *string2;  /* Source string */

 /*  Find the length of the first substring  *
  * in string1 of characters not in string2 */
size_t strcspn(string1, string2)
const char *string1;  /* Source string */
const char *string2;  /* Character set */

  /* Compare strings without regard to case */
int strcmpi(string1, string2)
int stricmp(string1, string2)
const char *string1;
```

const char *string2;

char *strdup(string) /* Duplicate string * /
const char *string; /* Source string */

Comments:

The **strcat, strchr, strcmp, strcmpi, stricmp, strcpy, strcspn** and **strdup** functions operate on null-ended 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 a string is copied or something is added to it.

The **strcat** function adds *string2* to *string1*, ends the resulting string with a null character, and returns a pointer to the concatenated string (*string1*).

The **strchr** function returns a pointer to the first occurrence of *c* converted to a character in *string*. The character *c* can be the null character (\0); the ending null character of *string* is included in the search. The function returns NULL if the character is not found.

The **strcmp** function compares *string1* and *string2* and returns a value indicating their relationship, as follows:

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* identical to *string2*
Greater than 0	*string1* greater than *string2*.

The **strcmpi** and **stricmp** functions are case-insensitive versions of **strcmp**. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison, so *string1* and *string2* are compared without regard to case.

The **strcpy** function copies *string2*, including the ending null character, to the location specified by *string1* and returns *string1*.

The **strcspn** function returns the index of the first character in *string1* that belongs to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that con-

sists entirely of characters not in *string2*. Ending null characters are not considered in the search. If *string1* begins with a character from *string2*, **strcspn** returns 0.

The **strdup** function reserves 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 it cannot reserve storage.

Example:

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

char string[100],template[100],*result,*newstring;
int numresult;
main()
{
   /* Construct the string "computer   *
    * program" using strcpy and strcat.*/
strcpy(string, "computer");
result = strcat(string, " program");
printf("Result = %s\n", result);

   /* Search a string for the occurrence of 'a' */
result = strchr(string, 'a');

   /* Determine whether a string is less than, *
    * greater than, or equal to another.       */

numresult = strcmp(string, template);

   /* Compare two strings without regard to case */
numresult = strcmpi("hello", "HELLO");
   /* Result is 0 */

   /* Make a copy of a string */
result = strcpy(template, string);

   /* Search for  a's, b's, or c's in a string */
strcpy(string,"xyzabbc");
numresult = strcspn(string, "abc");
   /* Result is 3 */

/* Make new string point to a duplicate of string */
newstring = strdup(string);
printf("The new string is %s\n", newstring);
}
```

Related Topics:

strncat, strncmp, strncpy, strrchr, strspn, strpbrk

strerror

Purpose:

Tests for system error and returns a pointer to the string containing the system error message.

Format:

```
  /* Required only for function declarations */
#include <string.h>

char *strerror(string)
char *string;  /* Your system error message */
int errno;  /* Error number */
int sys_nerr;  /* Number of system messages */
  /* Array of error messages */
char *sys_errlist[sys_nerr];
```

Comments:

If *string* is equal to NULL, the **strerror** function returns a pointer to a string containing the system error message for the last library call that produced an error. The newline character (**\n**) ends this string.

If *string* is not equal to NULL, **strerror** returns a pointer to a string containing:

- Your string message
- A colon
- A space
- The system error message for the last library call producing an error
- A newline character.

Your *string* message can be a maximum of 94 bytes long.

Unlike **perror**, **strerror** by itself does not print a message. To print the message returned by **strerror** to **stderr**, your program must have a **printf** statement similar to the following:

```
if ((access("datafile",2)) == -1)
      printf(strerror(NULL));
```

The compiler stores the error number in the variable *errno*, which you should declare at the external level. You get access to the system error messages through the variable *sys_errlist*, which is an array of messages ordered by the *errno* variable. **Strerror** gets access to the appropriate error message by using the *errno* value as an index to *sys_errlist*. The value of the variable *sys_nerr* is the maximum number of elements in the **sys_errlist** array.

To produce accurate results, call **strerror** immediately after a library routine returns with an error. Otherwise, subsequent calls might write over the *errno* value.

Note: DOS does not use some of the *errno* values listed in **errno.h**. See Appendix A, "Error Messages" for a list of *errno* values used on DOS and the corresponding error messages. The **strerror** function prints an empty string for an *errno* value not used under DOS.

Example:

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

int fh1, fh2;

fh1 = open("data1",O_RDONLY);
if (fh1 == -1)
   printf(strerror("open failed on input file"));

fh2 = open("data2",O_WRONLY|O_TRUNC|O_CREAT    ,
          S_IREAD|S_IWRITE);
if (fh2 == -1)
   printf(strerror("open failed on output file"));
```

Related Topics:

clearerr, ferror, perror

strlen

Purpose:

Returns the length of a *string*.

Format:

```
  /* Required for function declarations */
#include <string.h>

size_t strlen(string)
const char *string;  /* Null-ended string */
```

Comments:

The **strlen** function returns determines the length, in bytes, of *string*, not including the ending null character (\0).

The **strlen** function returns the *string* length. There is no error return.

Example:

The following example determines the length of the string to which *string* points.

```
#include <string.h>

char *string = "some space";
size_t result;
main()
{
result = strlen(string);   /* result = 10 */
}
```

Purpose:

Converts uppercase to lowercase in a null-ended *string*.

Format:

```
  /* Required for function declarations */
#include <string.h>
char *strlwr(string)
char *string;   /* String to convert */
```

Comments:

The **strlwr** function converts any uppercase letters in the given null-ended *string* to lowercase. Other characters are not affected.

The **strlwr** function returns a pointer to the converted *string*. There is no error return.

Example:

This example makes a copy in all-lowercase of the string "General Assembly", then prints the copy.

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

main()
{
    char *string="General Assembly";
    char *copy;

    copy=strlwr(strdup(string));
    printf("%s\n",copy);
}
```

Related Topics:

strupr

Purpose:

Operate on null-ended strings. The **strn***ccc* functions process only the *n* bytes specified.

Format:

```
  /* Required for function declarations */
#include <string.h>

  /*  Append n characters  *
   * of string2 to string1 */
char *strncat(string1, string2, n)
char *string1;  /* Destination string */
const char *string2;  /* Source string */
  /* Number of characters to append */
size_t n;

  /*   Compare first n    *
   * characters of strings */
int strncmp(string1, string2, n)
const char *string1;
const char *string2;
  /* Number of characters to compare */
size_t n;

  /*  Copy n characters   *
   * of string2 to string1 */
char *strncpy(string1, string2, n)     .
char *string1;  /* Destination string */
const char *string2;   /* Source string */
  /* Number of characters to copy */
size_t n;

  /* Compare first n characters of   *
   * strings without regard to case */
int strnicmp(string1, string2, n)
const char *string1;
const char *string2;
```

```
  /* Number of characters to compare */
size_t n;

  /* Initialize first n characters of string */
char *strnset(string, c, n)
char *string; /* String to be initialized */
int c;  /* Character setting */
size_t n;  /* Number of characters to set */
```

Comments:

The **strncat**, **strncmp**, **strncpy**, **strnicmp**, and **strnset** functions operate on, at most, the first *n* characters of null-ended strings.

The **strncat** function appends, at most, the first *n* characters of *string2* to *string1*, ends the resulting string with a null character (\0), and returns a pointer to the joined string (*string1*). If *n* is greater than the length of *string2*, the length of *string2* is used in place of *n*.

The **strncmp** function compares, at most, the first *n* characters of *string1* and *string2* and returns a value indicating the relationship between the substrings, as listed below.

Value	Meaning
Less than 0	*substring1* less than *substring2*
0	*substring1* equivalent to *substring2*
Greater than 0	*substring1* greater than *substring2*.

Strnicmp is a case-insensitive version of **strncmp**; **strnicmp** converts all alphabetic characters in the two strings *string1* and *string2* to lowercase before comparing them, so that the uppercase (capital) and lowercase forms of a letter are considered equivalent.

The **strncpy** function copies exactly *n* characters of *string2* to *string1* and returns *string1*. If *n* is less than the length of *string2*, a null character (\0) is *not* automatically appended to the copied string. If *n* is greater than the length of *string2*, the *string1* result is padded with null characters (\0) up to length *n*.

strncat - strnset

The **strnset** function sets at most the first *n* characters of *string* to *c* (converted to a character) and returns a pointer to the altered *string*. If *n* is greater than the length of *string*, the length of *string* is used in place of *n*.

Example:

This program demonstrates the uses of **strncat**, **strncmp**, **strnicmp**, and **strnset**.

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

char string[100] = "XYZabbc This is a string!";
char copy[100] = "This is a different string";
char *result;
char suffix[100]="this is even more string...";
int numresult;

main()
{
    /*  Combine strings with no more than  */
    /*  100 characters of suffix:          */
    printf("String before=%s\n",string);
    result=strncat(string,suffix,100);
    printf("string after=%s\n",string);

    /*  Determine ordering of two strings  */
    /*  but consider only 7 characters.    */
    strcpy(string,"programming");
    numresult=strncmp(string,"program",7);
    printf("%s is %s %s\n",string,
      numresult? numresult>0 ? "greater than"
      : "less than" : "equal to","program");
    /*  Copy at most 99 characters.        */
    printf("%s   %s\n",copy,string);
    result=strncpy(copy,string,99);
    copy[99]='\0';
    printf("%s   %s\n",copy,string);

    /*   set not more than 4 characters  */
    /*   of a string to be x             */
    result=strnset("computer",'x',4);
    printf("%s\n",result);
}
```

Related Topics:

strcat, strcmp, strcpy, strset

Purpose:

Finds the first occurrence in *string1* of any character from *string2*.

Format:

```
  /* Required for function declarations */
#include <string.h>

  /* Find any character from. *
   *   string2 in string1    */
char *strpbrk(string1, string2)
const char *string1;  /* Source string */
const char *string2;  /* Character set */
```

Comments:

The **strpbrk** function finds the first occurrence in *string1* of any character from *string2*. The ending null character (\0) is not included in the search.

The **strpbrk** function returns a pointer to the first occurrence of any character from *string2* in *string1*. A NULL pointer indicates that *string1* and *string2* have no characters in common.

Example:

The following example returns a pointer to the first occurrence in the array *string* of either the character **a** or the character **b**.

```
#include <string.h>

char string[100], *result;
     .
     .
     .
result = strpbrk(string, "ab");
```

Related Topics:

strchr, strrchr, strcspn

strrchr

Purpose:

Finds the last occurrence of the character *c* in *string2*.

Format:

```
  /* Required for function declarations */
#include <string.h>

const char *strrchr(string, c)
  /* Find last occurrence *
   *    of c in string    */
const char *string;  /* Searched string */
int c;  /* Character to locate */
```

Comments:

The **strrchr** function finds the last occurrence of *c* (converted to a character) in *string*. The ending null character (\0) is not included in the search.

The **strrchr** function returns a pointer to the last occurrence of *c* in *string*. A NULL pointer indicates that the given character is not found.

Example:

The following example searches a string for the last occurrence of the character **a** in the string.

```
#include <string.h>

char *result, string[ ] = "Baltimore";
      .
      .
      .
result = strrchr(string, 'a');
```

Related Topics:

strchr, strpbrk, strcspn

Purpose:

Reverses the order of characters in a given *string*.

Format:

```
 /* Required for function declarations */
#include <string.h>

char *strrev(string)
char *string;  /* String to be reversed */
```

Comments:

The **strrev** function reverses the order of the characters in the given *string*. The ending null character (\0) remains in place.

The **strrev** function returns a pointer to the altered *string*. There is no error-return value.

Example:

The following example determines if a string is a palindrome. A palindrome is a string that reads the same forwards and backwards.

```
#include <string.h>

char string[100];
int result;
      .
      .
      .

result = strcmp(string, strrev(strdup(string)));

/* If result equals 0, the string reads *
 * the same read forwards or backwards. */
```

Related Topics:

strcpy, strset

strset

Purpose:

Sets the characters of a given *string* to *c*.

Format:

```
  /* Required for function declarations */
#include <string.h>

char *strset(string, c)
char *string;  /* String to be set */
int c;  /* Character setting */
```

Comments:

The **strset** function sets all characters of *string* except the ending null character (\0) to *c* (converted to a character).

The **strset** function returns a pointer to the altered *string*. There is no error return.

Examples:

The following example sets a string to be all blanks.

```
#include <string.h>

char string[100], *result;
      .
      .
      .
result = strset(string, ' ');
```

Related Topics:

strnset

Purpose:

Returns the index of first character that does not belong in string.

Format:

```
  /* Required for function declarations */
#include <string.h>
size_t int strspn(string1, string2)
const char *string1;  /* Searched string */
const char *string2;  /* Character set */
```

Comments:

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 equal to the length of the initial substring of *string1* that consists entirely of characters from *string2*. The null character (\0) that ends *string2* is not considered in the matching process. If *string1* begins with a character not in *string2*, **strspn** returns 0.

The **strspn** function returns an integer value specifying the position of the first character in *string1* not in *string2*.

strspn

Example:

The following example returns a pointer to the first occurrence in the array *string* that is neither an **a**, **b**, nor **c**. Because the string in this example is **cabbage**, the value of the pointer is 5. You can use this function in text editors and word processors as a negative search function to locate special characters or control bytes.

```
#include <string.h>

char *string="cabbage";
int result;
        .
        .
        .
result = strspn(string, "abc");  /* result = 5 */
```

Related Topics:

strcspn

Purpose:

Returns a pointer to the first occurrence of a string in another string.

Format:

```
  /* Required only for a function declaration */
#include <string.h>

char *strstr(string1,string2)
const char *string1;  /* Searched string */
const char *string2;  /* String to search for */
```

Comments:

The **strstr** function returns a pointer to the first occurrence of *string2* in *string1*. The **strstr** function ignores the null character (**\0**) that ends *string2* in the matching process.

The **strstr** function returns a pointer to the first substring in *string1* equal to *string2*. If *string2* does not appear in *string1*, **strstr** returns NULL.

Example:

The following example locates the string **hay** in the string **needle in a haystack**.

```
#include <string.h>

char *string1 = "needle in a haystack";
char *string2 = "hay";
char *result;

result = strstr(string1,string2);
   /* Result = a pointer to "hay" */
```

Related Topics:

strcspn, strspn

strtod - strtol

Purpose:

Converts a character string to a double-precision value or a long-integer value.

Format:

```
#include <stdlib.h>

   /* Convert the string pointed  *
    *     to by nptr to double     */
double strtod(nptr,endptr)
const char *nptr;   /* Pointer to string */
char **endptr;   /* Pointer to end of scan */

   /* Convert string to long decimal integer *
    * equivalent of number given base */
long int strtol(nptr,endptr,base)
const char *nptr;
char **endptr;
int base;   /* Number base to use */
```

Comments:

The **strtod** and **strtol** functions convert a character string to a double-precision value or a long-integer value. The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified type. These functions stop reading the string at the first character that they cannot recognize as part of a number. This character can be the null character at the end of the string. With **strtol**, this ending character can also be the first numeric character greater than or equal to the *base*. If *endptr* is not NULL, **endptr* points to the character that stopped the scan.

Strtod expects *nptr* to point to a string with the following form:

[*white space*][*sign*][*digits*][.*digits*][d|D|e|E[*sign*]*digits*]

The first character that does not fit this form stops the scan.

Strtol expects *nptr* to point to a string with the following form:

[*white space*][*sign*][**0**| **0x**|**0X**][*digits*]]

If *base* is from 2 through 36, then it becomes the base of the number. If base is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

The letters from *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.

Strtod returns the value of the floating-point number, except when the representation causes an overflow. In those cases, it returns +/- HUGE_VAL.

Strtol returns the value represented in the string, except when the representation causes an overflow. In these cases, it returns LONG_MAX or LONG_MIN.

In both functions *errno* is set to ERANGE for the exceptional cases.

Example:

```
#include <stdlib.h>

main()
{
   char *string, *stopstring;
   double x;
   long l;
   int bs;

   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 = "10110134932";
   printf("string = %s\n", string);
   for (bs = 2; bs <= 8; bs *= 2) {
      l = strtol(string, &stopstring, bs);
      printf("strtol = %ld (base %d)\n", l, bs);
      printf("Stopped scan at %s\n\n", stopstring);
      }
}
```

Output:

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at This stopped it

string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at 34932

   strtol = 4423 (base 4)
   Stopped scan at 4932

   strtol = 2134108 (base 8)
   Stopped scan at 932
```

Related Topics:

atof, atol

Purpose:

Reads *string1* and *string2*.

Format:

```
  /* Required for function declarations */
#include <string.h>

  /* Find token in string1 */
char *strtok(string1, string2)
char *string1;  /* String containing token(s) */
const char *string2;  /* Set of delimiter characters */
```

Comments:

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* can be separated by one or more of the delimiters from *string2*. The tokens are broken out of *string1* by a series of calls to **strtok**.

In the first call to **strtok** for a given *string1*, **strtok** searches for the first token in *string1*, skipping over 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 can vary from call to call, so *string2* can take any 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-ended.

strtok

Example:

Using a loop, the following example gathers tokens, separated by blanks or commas, from a string until no tokens are left. After processing the tokens (not shown), the example returns the tokens **a**, **string**, **of**, and **tokens**. The next call to **strtok** returns NULL and the loop ends.

```
#include <string.h>
#include <stdio.h>
char *token;

char *string="a string, of, ,tokens ";
      .
      .
      .

token = strtok(string, " ,");

while (token != NULL) {
    /* Insert code to process the token here */
            .
            .
            .
   token = strtok(NULL," ,"); /* Get next token */
   }
```

Related Topics:

strcspn, strspn

Purpose:

Converts lowercase letters to uppercase.

Format:

```
  /* Required for function declarations */
#include <string.h>

char *strupr(string)
char *string;  /* String to be capitalized */
```

Comments:

The **strupr** function converts any lowercase letters in *string* to uppercase. Other characters are not affected.

The **strupr** function returns a pointer to the converted *string*. There is no error return.

Example:

This example makes a copy in all-uppercase of the string "DosCreateSem", then prints the copy.

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

main()
{
    char *string="DosCreateSem";
    char *copy;

    copy=strupr(strdup(string));
    printf("%s\n",copy);
}
```

Related Topics:

strlwr

swab

Purpose:

Copies and swaps adjacent bytes, then stores the result.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

void swab(source, destination, n)
  /* Data to be copied and  swapped */
char *source;
  /* Storage location for swapped data */
char *destination;
int n;   /* Number of bytes copied */
```

Comments:

The **swab** function copies *n* bytes from *source*, swaps each pair of adjacent bytes, and stores the result at *destination*. The integer *n* should be an even number to allow for swapping. The **swab** function is typically used to prepare binary data for transfer to a machine that uses a different byte order.

There is no return value.

Example:

The following example copies *n* bytes from one location to another, swapping each pair of adjacent bytes.

```
#include <stdlib.h>
#define NBYTES  1024

char from[NBYTES], to[NBYTES];

swab(from, to, NBYTES);
```

Related Topics:

fgetc, fputc

Purpose:

Passes a *string* to the command interpreter.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

int system(string)
const char *string;    /* Command to be run */
```

Comments:

The **system** function passes the given *string* to the command interpreter, which interprets and runs the string as a DOS command. In DOS mode the command interpreter is COMMAND.COM, while under OS/2 mode it is CMD.EXE. The **system** function refers to the COMSPEC and PATH environment variables to locate the appropriate command interpreter, which is used to run the *string* command.

The **system** function returns the value 0 if *string* successfully runs. A return value of -1 indicates an error, and *errno* is set to one of the following values:

Value	Meaning
E2BIG	The argument list for the command exceeds 128 bytes or the space required for the environment information exceeds 32K bytes.
ENOENT	COMMAND.COM or CMD.EXE cannot be found.
ENOEXEC	The command interpreter file has a non-valid format and is not executable.
ENOMEM	Not enough storage is available to run the command, or the available storage has been corrupted, or an incorrect block exists, indicating that the process making the call was not reserved properly.

Example:

```
#include <stdlib.h>
main()
{
int result;

/* The following statements append a copy of the DOS *
 * version number to a log file, then display it.     */

result = system("ver >>result.log");
result = system("type result.log");
}
```

Related Topics:

execl, execle, execlp, execv, execve, execvp, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

Note: If you use the system function to call a new command interpreter and do a DOS SET command, you will see a new environment variable called ";C_FILE_INFO". This variable exists only during the lifetime of a child program, such as the command interpreter during a SYSTEM call. ";C_FILE_INFO" contains binary-encoded information on open files.

Purpose:

Return tangent and hyperbolic tangent.

Format:

```
#include <math.h>
double tan(x)    /* Calculate tangent of x */
  /* Calculate hyperbolic  *
   * tangent of x  */
double tanh(x)
double x;    /* Angle in radians */
```

Comments:

The **tan** and **tanh** functions compute the tangent and hyperbolic tangent of *x*, respectively.

The **tan** function returns the tangent of *x*. If *x* is large, a partial loss of significance in the result can occur. In such cases, **tan** sets *errno* to ERANGE and generates a PLOSS error, but no message is printed. If *x* is so large that a total loss of significance occurs, **tan** prints a TLOSS error message to **stderr**, sets *errno* to ERANGE, and returns 0. For more information about ERANGE, PLOSS, and TLOSS, see Appendix A, "Error Messages", in this book.

The **tanh** function returns the hyperbolic tangent of *x*. There is no error return.

Example:

The following example computes *x* as the tangent of $\pi/4$ and *y* as the hyperbolic tangent of *x*.

```
#include <math.h>

double pi, x, y;

pi = 3.1415926535;
x = tan(pi/4.0);    /* x is 1.0 */
y = tanh(x);        /* y is .761594 */
```

Related Topics:

acos, asin, atan, atan2, cos, cosh, sin, sinh

Purpose:

Gets the current position of the file pointer.

Format:

```
  /* Required for function declarations */
#include <io.h>

long tell(handle)
int handle;  /* Handle referring to open file */
```

Comments:

The **tell** function gets the current position of any file pointer associated with *handle*. The position is the number of bytes from the beginning of the file.

The **tell** function returns the current position. A return value of -1L indicates an error, and *errno* is set to EBADF to indicate a non-valid file handle argument. On devices incapable of seeking (such as screens and printers), the return value is undefined.

tell

Example:

The following example opens the file DATA. After processing some statements (not shown), the example gets the current position of the file pointer, using **tell**. The example then assigns this value to *position*. After processing additional statements (not shown), the example uses **lseek** to return to the position stored in *position*.

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

int fh;
long position;

fh = open("data", O_RDONLY);
      .
      .
      .
position = tell(fh);
      .
      .
      .
lseek(fh, position, SEEK_SET);
```

Related Topics:

ftell, lseek

Purpose:

Returns the elapsed seconds since January 1, 1970.

Format:

```
   /* Required for function declarations */
#include <time.h>
time_t time(timeptr)
time_t *timeptr;   /* Storage location for time */
```

Comments:

The **time** function returns the number of seconds elapsed since 00:00:00 Greenwich Mean Time, January 1, 1970, according to the system clock. The return value is also stored in the location given by *timeptr*. If *timeptr* is NULL, the return value is not stored.

The **time** function returns the time in elapsed seconds. There is no error return.

Example:

This example gets the number of seconds elapsed since January 1, 1970 00:00 GMT and assigns it to ltime. It then uses the **ctime** function to convert the number of seconds to the current time. It prints a message giving the current date and time.

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

main()
{
    long ltime;
    time(&ltime);
    printf("The time is %s\n",
        ctime(&ltime));
}
```

Related Topics:

asctime, ftime, gmtime, localtime, utime

tmpfile

Purpose:

Creates a temporary file and returns a pointer to that file.

Format:

```
#include <stdio.h>
FILE *tmpfile( )    /*-Pointer to file structure */
```

Comments:

The **tmpfile** function creates a temporary file and returns a pointer to that file. If the file cannot be opened, **tmpfile** returns a NULL pointer.

The system deletes this temporary file when the file is closed, when the program ends, or when you call **rmtmp**, assuming that the current working directory does not change. The **tmpfile** function opens the temporary file in **"w+b"** mode.

Tmpfile returns a stream pointer, unless it cannot open the file, in which case it returns a NULL pointer.

The **tmpfile** routine uses the **tmpnam** routine to generate the name of the temporary file. For information on how the temporary file is named and which directory prefix is used, refer to the description of **tmpnam** in this chapter.

Example:

```
#include <stdio.h>

FILE *stream;
char tmpstring[ ] = "This is the string to be"
      "temporarily written";
main()

   {
   if((stream = tmpfile( )) == NULL)
      perror("Cannot make a temporary file");
   else
      fprintf(stream, "%s", tmpstring);
   }
```

Related Topics:

tmpnam, tempnam, rmtmp

tmpnam - tempnam

Purpose:

Produces a temporary filename in the same or another directory.

Format:

```
#include <stdio.h>

char *tmpnam(string)
char *string;    /* Pointer to temporary name */

char *tempnam(dir,prefix)
char *dir;
char *prefix;
```

Comments:

The **tmpnam** function produces a temporary filename that you can use as a temporary file. It stores this name in *string*. If *string* is NULL, **tmpnam** leaves the result in an internal static buffer. Any subsequent calls destroy this value. If *string* is not NULL, it points to an array of at least **L_tmpnam** bytes. The value of **L_tmpnam** is defined in **stdio.h**.

The **tempnam** function lets you create a temporary file in another directory. It uses *dir* as the directory to test for the existence of the name of the temporary file and *prefix* as the prefix to the filename. If *dir* is NULL, or if it is a non-existent directory, **tempnam** uses **P_tmpdir** in **stdio.h** for the directory. If **P_tmpdir** does not exist, \TMP is used. If this fails, **tempnam** uses the current working directory.

Tmpnam and **tempnam** both return a pointer to the temporary name, unless it is impossible to create this name or the name is not unique. If the name cannot be created, or if it already exists, **tmpnam** and **tempnam** return the value NULL.

Note: Because **tempnam** uses **malloc** to reserve space for the created filename, it is your responsibility to free this space when you no longer need it.

The character string that **tmpnam** creates consists of the path prefix defined by the **P_tmpdir** entry in **stdio.h**, followed followed by a sequence of the digit characters '0' through '9'; the numerical value of this string can range from 1 to 65535. Changing the definitions of **L_tmpnam** or **P_tmpdir** in **stdio.h** does not change the operation of **tmpnam**.

The **tempnam** function lets you create a temporary file in another directory. The *prefix* is the prefix to the filename. The **tempnam** function looks for the file with the given name in the following directories, listed in order of precedence:

Condition	Directory Used by tempnam
TMP environment variable is set and directory specified by **TMP** exists	Directory specified by **TMP**
TMP environment variable not set or directory specified by **TMP** does not exist	The *dir* argument to **tempnam**
The *dir* argument is NULL, or *dir* is name of nonexistent directory	**P_tmpdir** in **stdio.h**
P_tmpdir does not exist	The current working directory

If this is not successful, **tempnam** returns the value NULL.

tmpnam - tempnam

Example:

This program creates two temporary filenames: one in the current working directory, and one in A:\TMP with a prefix *stq*.

```
#include <stalib.h>

main()

   {
   char *name1, *name2;
   if ((name1 = tmpnam(NULL)) !=NULL)
      printf("%s is safe to use as a"
     "temporary file.\n", name1);
   else printf("Cannot create a unique filename\n");

   if((name2 = tempnam("a:\\tmp","stq")) != NULL)

      printf("%s is safe to use as a"
     "temporary file.\n", name2);
   else printf("Cannot create unique filename\n");
   }
```

Related Topics:

tmpfile

Purpose:

Convert a single character.

Format:

```
#include <ctype.h>

  /* Convert c to ASCII character */
int toascii(c)
  /* Convert c to lowercase if appropriate */
int tolower(c)
  /* Convert c to lowercase */
int _tolower(c)
  /* Convert c to uppercase if appropriate */
int toupper(c)
  /* Convert c to uppercase */
int _toupper(c)
int c;   /* Character to be converted */
```

Comments:

The **toascii**, **tolower**, **_tolower**, **toupper,** and **_toupper** macros convert a single character as specified.

The **toascii** function sets all but the low-order 7 bits of *c* to 0 so that the converted value represents a character in the ASCII character set. If *c* already represents an ASCII character, *c* is unchanged.

The **tolower** macro converts *c* to lower case if *c* represents an uppercase letter. Otherwise, *c* is unchanged.

The **_tolower** macro is a version of **tolower** to use only when *c* is known to be uppercase. The result of **_tolower** is undefined if *c* is not an uppercase letter.

The **toupper** macro converts *c* to uppercase if *c* represents a lowercase letter. Otherwise, *c* is unchanged.

toascii - _toupper

The **_toupper** macro is a version of **toupper** to use only when *c* is known to be lowercase. The result of **_toupper** is undefined if *c* is not a lowercase letter.

The **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper** return the possibly-converted character *c*. There is no error return.

Example:

This example prints four sets of characters. The first set is the ASCII characters having graphic images, which range from 0x21 through 0x7e. The second set takes integers 0x7f21 through 0x7f7e and applies the **toascii** function to them, yielding the same set of printable characters. The third set is the characters with all lowercase letters converted to uppercase. The fourth set is the characters with all uppercase letters converted to lowercase.

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

main()
{
 int      ch;

    printf("The graphic characters"
  " (0x21 to 0x7e) are:\n");
    for (ch = 33; ch <= 126; ch++)
  printf("%c",ch);
    printf("\nIntegers 0x7f21 to 0x7f7e"
" mapped into these characters are:\n");
     for (ch = 32545; ch <= 32638; ch++)
  printf("%c",toascii(ch));
    printf("\nWith lowercase converted"
     " to upper, these are:\n");
    for (ch = 33; ch <= 126; ch++)
  printf("%c",toupper(ch));
    printf("\nWith uppercase converted"
     " to lower, these are:\n");
    for (ch = 33; ch <= 126; ch++)
  printf("%c",tolower(ch));
}
```

Related Topics:

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

Note: These functions are used as macros. However, **tolower** and **toupper** are also used as functions because the macro versions do not correctly handle arguments with side effects. You can use the function versions 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**.

tzset

Purpose:

Assigns values to *daylight*, *timezone*, and *tzname*.

Format:

```
  /* Required for function declarations */
#include <time.h>

void tzset( )

  /* Daylight saving time flag */
int daylight;
  /* Difference in seconds from GMT */
long timezone;
  /* Three-letter time zone strings */
char *tzname [2];
```

Comments:

The **tzset** function uses the current setting of the environment variable TZ to assign values to three 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.

The value of the environment variable TZ must be a 3-letter time zone name, such as EST, followed by a (possibly) signed number giving the difference in hours between Greenwich Mean Time and local time. The number can be followed by a 3-letter daylight saving time zone, such as EDT. For example, "EST5EDT" represents a valid TZ value for the Eastern Standard Time zone.

When **tzset** is called, the difference in seconds between Greenwich Mean Time and local time is stored in *timezone*. The *daylight* variable is given a nonzero value if a daylight saving time zone is specified in the TZ setting. Otherwise, this variable is given the value 0. The **tzset** function assigns *tzname*[0] the string value of the 3-letter time zone name from the TZ setting; *tzname*[1] is assigned the string

value of the daylight saving time zone. If you omit the daylight saving time zone from the TZ setting, *tzname*[1] is assigned an empty string.

If TZ is not currently set, the default is "EST5EDT," which corresponds to the Eastern Time zone. The default for *daylight* is 1; for *timezone*, 18000; for *tzname*[0], "EST"; and for *tzname*[1], "EDT."

The functions **gmtime**, **localtime**, and **ftime** all retun a flag to indicate whether daylight savings time is in effect. That flag is true only if **daylight** is nonzero and the current date is during the period of the year when daylight savings time is in effect.

The functions **gmtime**, **localtime**, and **ftime**, all return a flag to indicate whether daylight savings time is in effect. The flag is true only if *daylight* is nonzero and the current date is during the period when daylight savings time is in effect.

There is no return value.

Example:

The following example uses the **putenv** function to give the TZ variable the value **EST5**. It then uses the **tzset** function to set the values of *daylight*, *timezone*, and *tzname* to the values implied in TZ: *daylight* becomes **0**; *timezone* becomes **18000**; *tzname*[0] becomes **EST**; *tzname*[1] becomes the empty string.

```
#include <time.h>
int daylight;
long timezone;
char *tzname[ ];
      .
      .
      .
putenv("TZ=EST5");
tzset();
```

Related Topics:

asctime, **ftime**, **localtime**, **gmtime**

ultoa

Purpose:

Converts digits to a null-ended character string and stores results.

Format:

```
  /* Required for function declarations */
#include <stdlib.h>

char ultoa(value, string, radix)
unsigned long value;  /* Number to convert */
char *string ;      /* String result */
int radix;  /* Base of value */
```

Comments:

The **ultoa** function converts the digits of *value* to a null-ended character string and stores the result in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value*; it must be in the range of 2 through 36.

The **ultoa** function returns a pointer to *string*. There is no error-return value.

Example:

The following example converts the digits of the unsigned long value 1344115000 to the hexadecimal representation 0x501d9138:

```
#include <stdlib.h>

int radix = 16;
char buffer[40];
char *p;
p = ultoa(1344115000L, buffer, radix);
```

Related Topics:

itoa, ltoa

Note: The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

Purpose:
Sets the file permission mask of the current process.

Format:

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

int umask(pmode)
int pmode;  /* Default permission setting */
```

Comments:
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 variable *pmode* contains one or both of the manifest constants S_WRITE and S_IREAD, defined in **sys\stat.h**. When both constants are given, they are joined with the bitwise OR operator |. The meanings of the *pmode* arguments are in the following table:

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If the write bit is set in the mask, any new files will be read-only. Under DOS all files are readable. It is not possible to give write-only permission. Thus, setting the read bit has no effect.

umask

The **umask** function returns the previous value of *pmode*. There is no error return.

Example:

The following example sets the permission mask to create a read-only file.

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

int oldmask;

oldmask = umask(S_IWRITE);
```

Related Topics:

chmod, **creat**, **mkdir**, **open**

Purpose:

Pushes character *c* back onto input *stream*.

Format:

```
#include <stdio.h>

int ungetc(c, stream)
int c; /* Character to be pushed */
FILE *stream; /* Pointer to file structure */
```

Comments:

The **ungetc** function pushes the character *c* back onto the given input *stream*. The *stream* must be buffered and open for reading. A subsequent read operation on the *stream* starts with *c*. An attempt to push EOF on 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** can be erased if an **fseek** or **rewind** function is called before the character is read from the *stream*.

The **ungetc** function returns the character argument *c*. The return value EOF indicates a failure to push back the specified character.

ungetc

Example:

In the following example, the **while** statement reads decimal digits from an input data stream, using arithmetic statements to compose the numeric values of the numbers as it reads them. When a non-digit character appears before the end of the file, **ungetc** replaces it in the input stream so that later input routines can process it.

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

FILE *stream;
int ch;
unsigned int result = 0;
      .
      .
      .
while ((ch = getc(stream)) != EOF && isdigit(ch))
   result = result * 10 + ch - '0';
if (ch != EOF)
   ungetc(ch,stream);
      /* Put the nondigit character back */
```

Related Topics:

getc, getchar, putc, putchar

Purpose:

Pushes character *c* back to the keyboard.

Format:

```
  /* Required for function declarations */
#include <conio.h>
int ungetch(c)
int c; /* Character to push back */
```

Comments:

The **ungetch** function pushes the character *c* back to the keyboard, causing *c* to be the next character read. The **ungetch** function fails if called more than once before the next read.

The **ungetch** function returns the character *c* if it is successful. A return value of EOF indicates an error.

ungetch

Example:

In this example, there is a token delimited by white-space characters. The example calls the **ungetch** function to replace the white-space character following the token. Other input routines can then process the delimiter.

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

main()
{
char buffer[100];
int count = 0;
int ch;
while (isspace(ch = getche()))
  /* Skip preceding white space */ ;
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 - end the token */
printf("\n%s\n", buffer);
}
```

Related Topics:

cscanf, getch, getche

Purpose:

Deletes a file.

Format:

```
  /* Required for function declarations */
#include <io.h>
int unlink(pathname)
  /* Path name of file to be removed */
char *pathname;
```

Comments:

The **unlink** function deletes the file specified by *pathname*.
The **unlink** function returns the value 0 if the file is successfully deleted. A return value of -1 indicates an error, and *errno* is set to one of the following values:

Value	Meaning
EACCESS	The pathname specifies a read-only file.
ENOENT	The file or pathname was not found, or the path name specifies a directory, or in OS/2 mode the filename was incorrect.

unlink

Example:

The following example deletes the file TMPFILE from the system or prints an error message if unable to delete it.

```
#include <io.h>
#include <stdio.h>
main()
{
int result;

result = unlink("tmpfile");
if (result == -1)
   perror("Cannot delete tmpfile");
}
```

Related Topics:

close

Purpose:

Sets modification time.

Format:

```
#include <sys\types.h>
#include <sys\utime.h>

int utime(pathname, times)
char *pathname;   /* File pathname */
  /* Pointer to stored time values */
struct utimbuf *times;
```

Comments:

The **utime** function sets the modification time for the file specified by *pathname*. The process must have write access to the file; otherwise, the time cannot be changed.

Although the **utimbuf** structure contains a field for access time, under DOS, only the modification time is set. If *times* is a NULL pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type **utimbuf**, defined in **sys\utime.h**. The modification time is set from the *modtime* field in this structure.

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
EACCESS	The pathname specifies a directory or read-only file.
EMFILE	There are too many open files. The file must be opened to change its modification time.
ENOENT	The file or pathname was not found, or in OS/2 mode the filename was incorrectly specified.

utime

Example:

The following example tries to set the last modification time of file \TMP\DATA to the current time. It prints an error message if it is unable to do so.

```
#include <sys\types.h>
#include <sys\utime.h>
#include <stdio.h>
#include <stdlib.h>

if (utime("\\tmp\\data", NULL) == -1)
   perror("utime failed");
```

Related Topics:

asctime, ctime, fstat, ftime, gmtime, localtime, stat, time

Purpose:

Accesses the arguments to a function when the function takes a fixed number of required arguments and a variable number of optional arguments.

Format:

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

  /* Macro to set arg-ptr to beginning *
   * of list. Variable-name is the *
   * identifier of the rightmost parameter *
   * in the function definition. */
void va_start(arg-ptr,variable-name)
  /* Macro which expands to */
  /* an expression having   *
   * the type and value of  *
   * the next element in the *
   *  argument list      */
type va_arg(arg-ptr,type)

  /* Function to set arg-ptr *
   * to the end of the list  */
void va_end(arg-ptr)
  /* Pointer to a list of arguments */
va_list arg-ptr;
  /* Type of argument to be retrieved */
  /* Parameter preceding first optional argument */
type variable-name;
```

Comments:

The **va_start**, **va_arg**, and **va_end** macros get to the arguments to a function when the function also takes a variable number of optional arguments. You declare required arguments as ordinary parameters to the function and get to the arguments through the parameter names.

The **stdarg.h** macros get to the optional arguments. The **va_start** macro sets the *arg-ptr* pointer to the first optional argument in the argument list. The *variable-name* argument may not be declared with register storage class. The argument *arg-ptr* must have a **va_list** type. The argument *variable-name* is the identifier immediately preceding the first optional argument in the argument list. Use the **va_start** macro before the **va_arg** macro.

The **va_arg** macro retrieves a value of the given *type* from the location given by *arg-ptr* and increases *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 retrieve arguments from the list any number of times within the function.

The **va_end** macro resets the pointer to NULL after all arguments have been retrieved.

The **va_arg** macro returns the current argument. The **va_start** and **va_end** macros do not return values.

Example:

The following example shows the passing of a variable number of arguments. The called function average () computes the mean of the integers passed to it. The main program calls the average function first with four arguments, then with five arguments.

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

main()
{

      int    n;

 /* Call with 4 arguments:  -1 ends the list  */
    n = average(2, 3, 4, -1);
    printf("Average is: %d\n",n);

/*  Call with 5 arguments:  -1 ends the list  */
    n = average(5, 7, 9, 11, -1);
    printf("Average is: %d\n",n);
}

average(first)
int first;
{
    int i = 0, count = 0, sum = 0;
    va_list arg_marker;

    va_start arg_marker, first);

    if (first ! = -1)
       sum = first;
    else
       return(0);

    count++;
    for (;
         (i = va_arg(arg(arg_marker,int)) > = 0;
         sum+=i, count++)
       ;

    return (sum0count);
{
```

Output:

Average is: 3 Average is: 8

Output:

```
Computer   99   2.718282    a
```

Related Topics:

vfprintf, vprintf, vsprintf

vfprintf-vsprintf

Purpose:

Prints stream data with a varying list of arguments.

Format:

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

int vprintf(format-string,arg-ptr)
const char *format-string; /* Format control */
  /* Pointer to a list of arguments */
va_list arg-ptr;

int vfprintf(stream,format-string,arg-ptr)
FILE *stream; /* Pointer to file structure */
const char *format-string;
va_list arg-ptr;

int vsprintf(target-string,format-string,arg-ptr)
  /* Storage buffer for output */
char *target-string;
const char *format-string;
va_list arg-ptr;
```

Comments:

The **vprintf**, **vfprintf**, and **vsprintf** functions are similar to their counterparts **printf**, **fprintf**, and **sprintf**. In **vprintf**, **vfprintf**, and **vsprintf**, *arg-ptr* points to a list of arguments whose number can vary during the running of the program. In contrast, **printf**, **fprintf**, and **sprintf** can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. The *arg-ptr* parameter has type *va-list*, which is defined in **stdarg.h**. The list of arguments is converted and output according to the format specifications in *format-string*.

If there is no error, **vprintf**, **vfprintf**, and **vsprintf** return the number of characters written to **stdout**, *stream*, or the target string, respectively. If there is an error, these functions return the value -1.

Example:

The following example shows the passing of a variable number of arguments.

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

main()
{
    char   *s = "Computer";
    int    i = 99;
    double fp = 2.71828182847;
    char   c = 'a';

    vout(1, s, i, fp, c);
}

vout(dum_var, ...)
int  dum_var;
{
    va_list arg_ptr;

    va_start(arg_ptr, dum_var);
    /* at least one variable must */
    /* be in the argument list    */
    vprintf("%s  %d  %f  %c\n", arg_ptr);
    va_end(arg_ptr);
}
```

Output:

```
Computer   99   2.718282    a
```

Related Topics:

printf, **fprintf**, **sprintf**

wait

Purpose:
The **wait** function delays the completion of a parent process until the end of one or more child processes.

Format:

#include <process.h>

int wait (*stat_loc*)
int **stat_loc*;

Comments:
The **wait** function delays a parent process until one of the immediate child processes stops. If all the child processes stop before wait is called, control returns immediately to the parent function. The **wait** function is available only under OS/2.

If not NULL, the *stat_loc* argument points to the location that holds information about the return status and return code of a child process that ends. The return status shows whether the child process stopped normally. If the child process stops normally, using a call to the OS/2 DOSEXIT function, the low-order and high-order bytes of the return status are as follows:

Byte	Contents
Low-order	0
High-order	Low-order byte of the code that the child process passed to DOSEXIT. The system calls DOSEXIT 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, if control from the child process fell through at the end of **main**, an unpredictable value.

If the child process stops for any other reason, the low-order and high-order bytes of the return status are as follows:

Byte	Contents
Low-order	Return-status code from OS/2 DOSCWAIT function:
High-order	0

Code	Meaning
1	Hard-error unexpected end
2	Trap operation
3	SIGTERM signal not intercepted.

If **wait** returns after unexpected end of a child process, it returns -1 to the parent process and sets *errno* to EINTR.

If **wait** returns after a normal end of a child process, it returns the process identifier of the child process to the parent process.

Otherwise, **wait** returns immediately with a value of -1. In this case, *errno* is set to ECHILD, indicating that no child processes exist for the particular process.

Example:

The following example creates a new process called CHILD.EXE, specifying NO_WAIT when the child is called. The parent calls wait() and waits for the child to stop running. The parent then displays the child's status word in hexidecimal.

```
#include <stdio.h>
#include <process.h>
int stat_child;

main()
{
    int i,result;
    result=spawnl(P_NOWAIT,"child.exe",
      "child.exe",NULL);
    /*  Display error status message, if any  */

     if ((i=wait(&stat_child))==-1)
       perror();
     else
       printf("child process %d is running\n",i);
    /*  Display the word returned when child ended.  */

    printf("child process was &*X\n",stat_child);
}
```

Related Topics:

cwait, exit, _exit, spawnl, spawnle, spawnlp, spawnlpe, spawnp, spawnv, spawnve, spawnvp, spawnvpe

Purpose:

Writes from buffer to file.

Format:

```
   /* Required for function declarations */
#include <io.h>

int write(handle, buffer, count)
int handle;  /* Handle referring to open file */
char *buffer;  /* Data to be written */
unsigned int count;  /* Number of bytes */
```

Comments:

The **write** function writes *count* bytes from *buffer* into the file associated with *handle*. The write operation begins at the current position of any file pointer associated with the given file. If the file is open for adding, the operation begins at the current end of the file. After the write operation, the file pointer (if any) is increased by the number of bytes actually written.

The **write** function returns the number of bytes actually written. The return value may be positive but less than *count* (for example, when running out of space on a disk before *count* bytes are written). A return value of -1 indicates an error, and *errno* is set to one of the following values.

Value	Meaning
EBADF	The file handle is non-valid or the file is not open for writing.
ENOSPC	No space is left on the device.

If you are writing more than 32K bytes to a file, the return value should be of the type **unsigned int**. The maximum number of bytes you can write to a file is 65534, because 65535 (0xFFFF) is indistinguishable from −1 and causes the return of an error.

If the given file was 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.

Example:

The following example writes the contents of the character array *buffer* to the output file whose handle is *fh*. The length of the buffer is *BUFSIZ*.

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

int fh, byteswritten;
unsigned int nbytes = BUFSIZ;
char buffer[BUFSIZ];
      .
      .
      .
byteswritten = write(fh, buffer, nbytes);
```

Related Topics:

cwait, exit, _exit, fwrite, open, read, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe

Note: When you write to files opened in text mode, a Ctrl+Z character is treated as the logical end of file. When you write to a device, a Ctrl+Z character in the buffer causes output to stop.

Appendix A. Error Messages

This appendix lists the error messages you might find as you develop a program and gives a brief description of the action required to correct the error. The first section lists run-time errors that you may find when you run your program.

The remaining sections describe error messages produced by the following programs:

- IBM C/2
- The IBM Linker
- The CodeView symbolic debugger
- The IBM LIB library management utility
- The EXEMOD header modification utility
- The MAKE program maintenance utility
- The ***errno*** variable and math routines.

Run-Time Library Error Messages

Run-time error messages are divided into four categories:

1. Error messages generated by the run-time library to notify you of serious errors.
2. Floating-point exceptions generated by the Numeric Coprocessor or the NPX emulator.
3. Error messages generated by calls in the program to error-handling routines in the C run-time library (the **abort, assert,** and **perror** routines). These routines print an error message to the standard error data stream (**stderr**) whenever the program calls the given routine. For a description of these routines, see Chapter 5 "Library Routines" in this book.

4. Error messages generated by calls to math routines in the C run-time library. On an error, the math routines return an error value or print a message to the standard error data stream. See Chapter 5, "Library Routines" for a description of the math routines.

When your program has serious errors, the system can generate the following messages at run time:

R6000: stack overflow

Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The system stops the program with an exit status of 255. To correct the problem, recompile using the **/F** option of the CL command, or relink using the linker /STACK option to reserve a large stack or modify the stack information in the executable file header by using the EXEMOD program.

R6001: null pointer assignment

The contents of the NULL segment changed as the program ran. The NULL segment is a special location in low storage that is not normally used. If the contents of the NULL segment change during the running of a program, the program has written to this area, usually by an inadvertent assignment through a null pointer. Your program can contain null pointers without generating this message; the message appears only when you get access to a storage location through the null pointer.

This error does not cause your program to stop; the system prints the error message following the normal end of the program.

This message reflects a potentially serious error in your program. Although a program that produces this error can appear to run correctly, it is likely to cause problems in the future and might fail to run in a different operating environment.

R6002: floating point not loaded

Your program needs the floating-point library, but that library was not loaded. This error stops the program with an exit status of 255. This error occurs in three situations:

1. A format string for one of the routines in the **printf** or **scanf** family contains a floating-point format specification, and there are no floating-point values or variables in the program. The C compiler tries to minimize the size of the program by

loading floating-point support only when necessary. It does not detect floating-point format specifications within format strings and, consequently, does not load the necessary floating-point routines. To correct this error, use a floating-point argument that corresponds to the floating-point format specification. This causes the C compiler to load floating-point support.

2. You specified *x*LIBFP.LIB or *x*LIBFA.LIB (where *x* is S, M, L, C, or H depending on the storage model) after *x*LIBC.LIB in the linking stage. You must relink the program with the correct library specification.

3. The program uses floating point and is compiled and linked with options that require a numeric coprocessor (-FPi87, for example), but is run on a machine that does not have a numeric coprocessor. You should either recompile with switch **/FPi**, relink with emulator library EM.LIB, or install a coprocessor.

R6003: integer divide by 0
An attempt was made to divide an integer by 0, giving an undefined result.

R6004: DOS 2.00 or later required
IBM C/2 cannot run on versions of DOS prior to 2.00

R6005: not enough memory on exec

R6006: bad format on exec

R6007: bad environment on exec
Errors R6005 through R6007 occur when a child process spawned by one of the **exec** library routines fails, and DOS was unable to return control to the parent process.

R6008: not enough space for arguments
See explanation under error R6009

R6009: not enough space for environment
Error R6008 and R6009 both occur at start-up if there is enough memory to load the program, but not enough room for the **argv** and/or **envp** vectors. To avoid this problem, you can rewrite the **_setargv** or **_setenvp** routines.

Floating-Point Exceptions

The error messages listed below correspond to exceptions produced by the numeric coprocessor. These messages appear with **error 2100: Floating point error,** described in the previous section. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions.

When you use the default floating-point control word settings in C, the following exceptions are masked and do not occur:

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Inexact	Exception masked.

The following errors do not occur with code that IBM C/2 produces or code provided in the IBM C/2 run-time library:

- Square root
- Stack underflow
- Unemulated.

The floating-point exceptions have this format:

runtime-time error M61*xx* : MATH
-floating-point error: *message text*

The following list describes the floating-point exceptions:

M6101: invalid
The operation is a non-valid operation. Usually this message appears when an operation tries to operate on NANs or infinities.

M6102: denormal
The operation produced a very small floating-point number, which might no longer be correct due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on.

M6103: divide by 0

The operation tried to divide by zero.

M6104: overflow

The operation produced an overflow in floating-point operation.

M6105: underflow

The operation produced an underflow in a floating-point operation. An underflow is normally masked so that the operation yields the result 0.0.

M6106: inexact

Loss of precision occurred in a floating-point operation. This exception is normally masked, because almost any floating-point operation can cause loss of precision.

M6107: unemulated

An attempt was made to run a floating-point instruction not supported by the emulator or a non-valid floating point instruction.

M6108: square root

The operand in a square root operation was negative.

Note: The **sqrt** function in the C run-time library checks the argument before performing the operation and returns an error value if the operand is negative. See Chapter 5, "Library Routines" for details on **sqrt.**

M6110: stack overflow

A floating-point expression has used too many stack levels on the numeric coprocessor or emulator. (Stack overflow exceptions are trapped up to a limit of seven additional levels beyond the eight levels normally supported by the numeric coprocessor.)

M6111: stack underflow

A floating-point operation resulted in a stack underflow on the numeric coprocessor or the emulator.

Run-Time Limits

The following table summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message informs you of the problem.

Program Limits at Run Time

Program	Item Description	Limit
Files	Maximum file size	2^{32}-1 bytes (4 gigabytes)
	Maximum number of open files (streams)[1]	20 for DOS 40 for OS/2
Command Line	Maximum number of characters (including program name)	128
Environment Table	Maximum size	32K

[1] Five streams are opened automatically (**stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**), leaving 15 available for the program to open.

Compiler Error Messages

The error messages produced by IBM C/2 fall into five categories:

- Warning messages
- Fatal error messages
- Error messages during compiling
- Command line messages
- Compiler internal error messages.

Warning messages are informational only; they do not prevent compiling and linking. You can control the level of warnings generated by the compiler by using the **/W** option, described in the *IBM C/2 Compile, Link, and Run* book. The list of warning messages includes a number for each message indicating the minimum level that must be set for the message to appear.

Fatal error messages indicate severe problems, those that prevent the compiler from processing your program. After printing out a message about the fatal error, the compiler stops without producing an object file or checking for further errors.

Error messages during compiling identify actual program errors. No object file is produced for a source file that has such errors. When the compiler finds a nonfatal program error, it tries to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Command line messages give you information about non-valid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler stops processing.

Compiler internal error messages indicate errors on the part of the compiler instead of an error in your program. The following messages are compiler internal error messages. No matter what your source program contains, these messages should not appear. If they do, please report the condition to your authorized IBM dealer. Although these errors are not the fault of your program, you will prob-

ably want to rearrange your code so that the program can be compiled.

C1000: UNKNOWN FATAL ERROR

C2000: UNKNOWN ERROR

An unforeseen error condition has been detected by the compiler.

C1001: Internal Compiler Error
(compiler file '<name>',line <n>)

The compiler performs internal consistency checks during compiling. This message indicates that the consistency check failed and the compiler cannot continue operation.

C4000: UNKNOWN WARNING

Error messages in the warning, fatal, and compiling error message categories have the same basic form:

filename (*linenumber*) : *msg-code error-number message*

The parts of the error message are as follows:

filename The name of the source file being compiled.

linenumber The line of the file containing the error.

msg-code The message code consists of two parts:

1. An initial letter which identifies the component that is reporting the error.
2. A single digit following the letter indicates the severity of the error.

The form of the message code with a number is:

<L> <N> <###>

Letter	Error Type
C	C Compiler
D	CL/CC driver
M	Math runtime errors
R	General runtime errors

Number	Error Type
1	Fatal Error
2	Error
4	Warning
6	Runtime

The <###> is the 3-digit error number within the category

error-number The number associated with the error

message A self-explanatory description of the error or warning. A command line error message gives a message about the command line; it does not contain references to line numbers and filenames.

The messages for each category are listed below in numerical order, along with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number.

"Compiler Limits" in this appendix summarizes the limits, such as the maximum size of a macro definition, that the IBM C/2 compiler imposes.

Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it stops after printing the error message.

C1001: Internal Compiler Error
(compiler file '<name>',line <n>)
The compiler has detected an internal error. Please report this error to your authorized IBM dealer. Include the compiler filename and line number information.

C1002: out of heap space
The compiler has run out of dynamic storage space. This usually means that your program has many symbols and complex expressions. To correct the problem, break down the file into several smaller source files.

C1003: error count exceeds *n*; stopping compilation
Errors in the program are too numerous or too severe to allow recovery, and the compiler must stop.

C1004: unexpected EOF
This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file.

C1006: write error on compiler intermediate file
The compiler is unable to create the intermediate files used in the compiling process. The exact reason is unknown.

C1007: unrecognized flag '*string*' in 'option'
The given *string* in the command line *option* is not a valid option.

C1009: compiler limit : possibly a recursively defined macro
The expansion of a macro exceeds the available space. Check to see whether the macro is recursively defined or if the expanded text is too large.

C1010: compiler limit : macro expansion too big
The expansion of a macro exceeds the available space.

C1012: bad parenthesis nesting - missing '*character*'
The parentheses in a preprocessor directive are not matched. The *character* is either (or).

C1013: cannot open source file '*pfilename*'

The given source file cannot be opened. Check to make sure you have given the correct pathname for the file. The system may have run out of file handles; you should have a line FILES=20 in your CONFIG.SYS file.

C1014: too many include files

Nesting of **#include** directives exceeds the limit of 10 levels.

C1015: cannot open include file '*filename*'

The given file cannot be opened. Check to make sure your INCLUDE environment variable is correct. The system may have run out of file handles; you should have a line FILES=20 in your CONFIG.SYS file. If your include files are shared, they should be read-only.

C1016: #if[n]def expected an identifier

You must specify an identifier with the **#ifdef** and **#ifndef** directives.

C1017: invalid integer constant expression

The expression in an **#if** directive must evaluate to a constant.

C1018: unexpected '#elif'

The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C1019: unexpected '#else'

The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C1020: unexpected '#endif'

An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

C1021: bad preprocessor command '*string***'.**

The characters following the number sign (#) do not form a preprocessor directive.

C1022: expected '#endif'

An **#if**, **#ifdef**, or **#ifndef** directive does not end with an **#endif** directive.

C1026: parser stack overflow, please simplify your program

Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, simplify your program.

C1027: DGROUP data allocation exceeds 64K

Large, compact, or huge model allocation of variables to the default segment exceeds 64K bytes. Use the /GT option to move items into separate segments.

C1032: cannot open listing file *'filename'*

The filename or pathname given for the listing file is not valid.

C1033: cannot open assembly language output file *'filename'*

The filename or pathname given for the assembly language output file is not valid.

C1034: cannot open source file *'filename'*

The filename or pathname given for the source file is not valid.

C1035: expression too complex, please simplify

The compiler cannot produce code for a complex expression. Break the expression into simpler subexpressions and recompile.

C1036: cannot open source-listing file *'filename'*

The filename or pathname given for the source file is not valid.

C1037: cannot open object file *'filename'*

The filename or pathname given for the source file is not valid.

C1039: unrecoverable heap overflow in Pass 3

The post-optimizer compiler pass has overflowed the heap and cannot continue. Try recompiling with the /Od option or breaking up the function containing the line causing the error.

C1040: unexpected EOF in source file *'filename'*

The compiler detected an unexpected end-of-file while creating a source listing or mingled a source/object listing. The probable cause is a source file edited during compiling. This error most likely occurs on a multi-tasking system where the compiling can be done as a background process.

C1041: cannot open compiler intermediate file -no more files

The compiler is unable to create intermediate files used in the compiling process because no more file handles are available. This can usually be corrected by changing the **files**= line in the CONFIG.SYS file to allow a larger number of open files (20 is the recommended setting).

C1042: cannot open compiler intermediate file - no such file or directory

The compiler is unable to create intermediate files used in the compiling process because the TMP environment variable is set to a non-valid directory or path.

C1043: cannot open compiler intermediate file

The compiler is unable to create intermediate files used in the compiling process. The exact reason is unknown.

C1044: out of disk space for compiler intermediate file

The compiler is unable to create intermediate files used in the compiling process because no more space is available. To correct the problem, make more space available on the disk and recompile.

C1045: floating point overflow

The compiler has produced a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:

```
float fp_val = 1.0e100
```

In this case, the double-precision constant 1.0e100 exceeds the maximum allowable value for a floating-point data item.

C1047: too many *option* flags, '*string*'

There are too many occurrences of the given *option*; *string* contains the occurrence of the *option* causing the error.

C1048: Unknown option '*character*' in '*optionstring*'

The specified *character* is not a valid letter for *optionstring*.

C1049: invalid numerical argument '*string*'

A numerical argument was expected instead of *string*.

C1050: '*segname*': code segment too large

The code generated for the given segment exceeded 64K.

C1051: program too complex

Simplify your program.

C1052: too many #if/#ifdefs

Your #if or #ifdef directives are nested more than 32 levels deep.

C1053: compiler limit : struct/union nesting

Nesting of structure and union definitions are limited to ten levels of nesting.

C1054: compiler limit : initializers too deeply nested

The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 through 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

C1056: compiler limit : out of macro expansion space

The expansion of a macro (often nested macros and large actual parameters) has used up the available space in the macro expansion buffer.

C1057: unexpected EOF in macro expansion

The preprocessor encountered an end-of-file while collecting the actual arguments for a macro expansion. This is usually caused by a missing) closing the macro argument list.

C1059: out of near heap space

Program too large (too many symbols) and the compiler cannot allocate space in the near heap.

C1060: out of far heap space

Program too large (too many symbols) and the compiler cannot allocate space in the far heap. Try removing other memory resident programs to create extra memory space. If your machine is a network server, you could reconfigure it so that it does not use network software during compiling.

C1062: error writing to preprocessor output file

A **-P** option was entered to create a preprocessor listing file. However, there is no available space on the output directory.

Error Messages During Compiling

The messages listed below indicate that your program has errors. When the compiler finds any of the errors listed in this section, it continues parsing the program, if possible, and puts out additional error messages. However, no object file is produced.

C2000: UNKNOWN ERROR
The compiler has detected an unforeseen error condition. Please report this error to your authorized IBM dealer.

C2001: newline in constant
A newline character in a character or string constant must be preceded by the backslash escape character (\).

C2002: out of macro actual parameter space
Arguments to preprocessor macros cannot exceed 256 bytes.

C2003: expected 'defined id'
An #if directive has a syntax error.

C2004: expected 'defined(id)'
An **#if** directive has a syntax error.

C2005: #line expected a line number
A **#line** directive lacks the mandatory line number specification.

C2006: #include expected a filename
An **#include** directive lacks the mandatory filename specification.

C2007: #define syntax
A **#define** directive has a syntax error.

C2008: '*c*' : unexpected in macro definition
The character *c* is misused in a macro definition.

C2009: reuse of macro formal '*identifier*'
The parameter list in a macro definition contains two occurrences of the same identifier.

C2010: '*c*' : unexpected in formal list
The character *c* is misused in the list of formal parameters for a macro definition.

C2011: '*identifier*' : definition too big
Macro definitions cannot exceed 512 bytes.

C2012: missing name following '<'
An **#include** directive lacks the mandatory filename specification.

C2013: missing '>'
The closing angle bracket '>' is missing from an **#include** directive.

C2014: preprocessor command must start as first non-whitespace
Non-whitespace characters appear before the number sign # of a preprocessor directive on the same line.

C2015: too many chars in constant
A character constant is limited to a single character or escape sequence. (Multi-character character constants are not supported.)

C2016: no closing single quote
Backslash escape character (\) must precede a newline character in a character constant.

C2017: illegal escape sequence
The characters after the escape character (\) do not form a valid escape sequence.

C2018: unknown character '0x*n*'
The given hexadecimal number does not correspond to a character.

C2019: expected preprocessor command, found '*c*'
The character following a number sign (#) is not the first letter of a preprocessor directive.

C2020: bad octal number '*n*'.
The character *n* is not a valid octal digit.

C2021: expected exponent value, not '*n*'
The exponent of a floating-point constant is not a valid number.

C2022: '*n*' : too big for char
The number *n* is too large to be represented as a character.

C2023: divide by 0
The second operand in a division operation (*/*) evaluates to zero, giving undefined results.

C2024: mod by 0
The second operand in a remainder operation (**%**) evaluates to zero, giving undefined results.

C2025: '*identifier*' : enum/struct/union type redefinition
The given *identifier* has already been used for an enumeration, structure, or union tag.

C2026: '*identifier*' : member of enum redefinition
The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

C2028: struct/union member needs to be inside a struct/union
Structure and union members must be declared within the structure or union.

C2029: '*identifier*' : bit-fields only allowed in structs
Only structure types can contain bit-fields.

C2030: struct/union member redefinition
The same identifier was used for more than one structure or union member.

C2031: '*identifier*' : function cannot be struct/union member
A function cannot be a member of a structure. Use a pointer to a function instead.

C2032: '*identifier*' : base type with near/far not allowed
Declarations of structure and union members cannot use the **near** and **far** keywords.

C2033: '*identifier*' : bit-field cannot have indirection
The bit field is declared as pointer, *, which is not allowed.

C2034: '*identifier*' : bit-field type too small for number of bits
The number of bits specified in the bit field declaration exceeds the number of bits in the given **unsigned** type.

C2035: enum/struct/union *'identifier'* : unknown size
A member of a structure or union has an undefined size.

C2036: left of '->*identifier*' must have struct/union type
The expression before member selection operator '->' is not a pointer to a structure or union type, or the expression before member selection operator '.' does not evaluate to a structure or union.

C2037: left of '->' specifies undefined struct/union '*identifier*'
The expression before member selection operator '->' or '.' identifies a structure or union type that is not defined.

C2038: '*identifier*' : not struct/union member
The given *identifier* is used in a context that requires a structure or union member.

C2039: '->' requires struct/union pointer
The expression before member selection operator '->' is not a pointer to a structure or union.

C2040: '.' requires struct/union name
The expression before member selection operator '.' is not the name of a structure or union.

C2042: signed/unsigned mutually exclusive
You may declare an identifier type as *signed* or *unsigned*, but not both.

C2043: illegal break
A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

C2044: illegal continue
A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

C2045: '*identifier*' : label redefined
The given *identifier* appears before more than one statement in the same function.

C2046: illegal case
The **case** keyword can appear only within a **switch** statement.

C2047: illegal default
The **default** keyword can appear only within a **switch** statement.

C2048: more than one default
A **switch** statement contains too many **default** labels. Only one is allowed.

C2050: non-integral switch expression
Switch expressions must be integers.

C2051: case expression not constant
Case expressions must be integer constants.

C2052: case expression not integral
Case expressions must be integer constants.

C2053: case value '*n*' already used
The decimal equivalent of case value *n* has already been used in this **switch** statement, where *n* is an integer constant.

C2054: expected '(' to follow '*identifier*'
The context requires parentheses after the function *identifier*.

C2055: expected formal parameter list, not a type list
An argument type list appears in a function definition where a formal parameter list should appear.

C2056: illegal expression
An expression is illegal because of a previous error. The previous error did not produce an error message.

C2057: expected constant expression
The context requires a constant expression.

C2058: constant expression is not integral
The context requires an integer constant expression.

C2059: syntax error : '*token*'
The given *token* caused a syntax error.

C2060: syntax error : EOF
The end of the file was found unexpectedly, causing a syntax error.

C2061: syntax error : identifier '*identifier*"
The given *identifier* caused a syntax error.

C2062: type '*identifier*' unexpected
The given type is misused.

C2063: '*identifier*' : not a function
The given *identifier* was not declared as a function, but an attempt was made to use it as a function.

C2064: term does not evaluate to a function
An attempt is made to call a function through an expression that does not evaluate to a function pointer.

C2065: '*identifier*' : undefined
The given *identifier* is not defined.

C2066: cast to function returning . . . is illegal
An object cannot be cast to a function type.

C2067: cast to array type is illegal
An object cannot be cast to an array type.

C2068: illegal cast
A type used in a cast operation is not a legal type.

C2069: cast of 'void' term to non-void
The **void** type cannot be cast to any other type.

C2070: illegal sizeof operand
The operand of a **sizeof** expression must be an identifier or a type name.

C2071: '*class*' : bad storage class
The given storage *class* cannot be used in this context.

C2072: '*identifier*' : initialization of a function
Functions cannot be initialized.

C2073: '*identifier*' : cannot initialize array in function
Arrays can be initialized only at the external level.

C2074: '*identifier*' cannot initialize struct/union in function
Structures and unions can be initialized only at the external level.

C2075: '*identifier*' : array initialization needs curly braces
The braces { } around an array initializer are missing

C2076: struct/union initialization needs curly braces
The braces { } around a structure or union initializer are missing.

C2077: non-integral field initializer '*identifier*'
An attempt is made to initialize a bit field member of a structure with a non-integer value.

C2078: too many initializers
The number of initializers exceeds the number of objects to be initialized.

C2079: '*variable*' uses an undefined struct/union identifier
The given *variable* is declared as a structure or union type identifier that has not been defined.

C2082: redefinition of formal parameter '*identifier*'
A formal parameter to a function is redeclared within the function body.

C2083: array '*identifier*' already has a size
The dimensions of the given array have already been declared.

C2084: function '*identifier*' already has a body
The given function has already been defined.

C2085: '*identifier*' : not in formal parameter list
The given identifier was declared in the list of argument declarations for a function, but was not listed in the formal parameter list in the function header.

C2086: '*identifier*' : redefinition
The given *identifier* was defined more than once.

C2087: '*identifier*' : missing subscript
To refer to an element of an array, you must use a subscript.

C2088: use of undefined enum struct/union '*identifier*'
The *identifier* refers to a structure, enumeration, or union type that is not defined.

C2089: typedef specifies a near/far function
The **near** or **far** keyword is used in a **typedef** declaration.

C2090: function returns array
A function cannot return an array. It can return a pointer to an array.

C2091: function returns function
A function cannot return a function. It can return a pointer to a function.

C2092: array element type cannot be function
Arrays of functions are not allowed.

C2093: cannot initialize a static or struct with address of automatic vars
You tried to initialize a static pointer to the address of a local variable.

C2094: label '*identifier*' was undefined
The function does not contain a statement labeled with the *identifier*.

C2095: parameter has type void
Formal parameters and arguments to functions cannot have **void** type.

C2096: struct/union comparison illegal
You cannot compare two structures or unions. You can, however, compare individual members of structure and unions.

C2097: illegal initialization
An initialization is illegal because of a previous error. The previous error might not have produced an error message.

C2098: non-address expression
An attempt was made to initialize an item that is not an lvalue.

C2099: non-constant offset
An initializer uses a non-constant offset.

C2100: illegal indirection
Indirection operator * was applied to a non-pointer value.

C2101: '&' on constant
Only variables and functions can have their address taken.

C2102: '&' requires lvalue
Address-of operator **&** can be applied only to *lvalue* expressions.

C2103: '&' on register variable
Register variables cannot have their address taken.

C2104: '&' on bit-field
Bit-fields cannot have their address taken.

C2105: '*operator*' needs lvalue
The *operator* must have an lvalue operand.

C2106: '*operator*' : left operand must be lvalue
The left operand of the *operator* must be an lvalue.

C2107: illegal index, indirection not allowed
A subscript was applied to an expression that does not evaluate to a pointer.

C2108: non-integral index
Only integer expressions are allowed in array subscripts.

C2109: subscript on non-array
A subscript was used on a variable that is not an array.

C2110: '+' : 2 pointers
Two pointers cannot be added.

C2111: pointer + non-integral value
Only integer values can be added to pointers.

C2112: illegal pointer subtraction
Only pointers that point to the same type can be subtracted.

C2113: '-' : right operand pointer

The right-hand operand in a subtraction operation (-) is a pointer, but the left-hand operand is not.

C2114: '*operator*' : pointer on left; needs integral right

The left operand of the *operator* is a pointer; the right operand must be an integer value.

C2115: '*identifier*' : incompatible types

An expression contains types that are not compatible.

C2116: '*operator*' : bad left *or* right operand

The specified operand of the *operator* is an illegal value.

C2117: '*operator*' : illegal for struct/union

Structure and union type values are not allowed with the *operator*.

C2118: negative subscript

A value defining an array size was negative.

C2119: 'typedefs' both define indirection

Two **typedef** types are used to declare an item and both **typedef** types have indirection. For example, the declaration of *pshint;* in the following example is illegal:

```
typedef int *P_INT;
typedef short *P_SHORT;
  /* This declaration is illegal */
P_SHORT P_INT pshint;
```

C2120: 'void' illegal with all types

The **void** type cannot be used in declarations with other types.

C2121: typedef specifies different enum

Two different enumeration types defined with **typedef** are used to declare an item, and the enumeration types are different.

C2122: typedef specifies different struct

Two structure types defined with **typedef** are used to declare an item, and the structure types are different.

C2123: typedef specifies different union

Two union types defined with **typedef** are used to declare an item, and the union types are different.

C2125: '*identifier*': allocation exceeds 64K

The given item exceeds the limit of 64K bytes. The only items that are allowed to exceed 64K bytes are **huge** arrays.

C2126: '*identifier*': automatic allocation exceeds *size*
The space allocated for the local variables of a function exceeds the given limit.

C2127: parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeds the limit of 32K bytes.

C2128: '*identifier*' huge array cannot be aligned to segment boundary
The given array violates one of the restrictions imposed on huge arrays. See *IBM C/2 Compile, Link, and Run*, Chapter 3.

C2129: static function '*identifier*' not found
A forward reference was made to a missing static procedure.

C2130: #line expected a string containing the filename
A **#line** directive is missing a filename.

C2131: attributes specify more than one near/far/huge
More than one **near**, **far**, or **huge** attribute was applied to an item, as in the following example:

```
typedef int near NINT;
NINT far a;      /* Illegal */
```

C2132: syntax error: unexpected identifier
The given identifier caused a syntax error.

C2133: array '*identifier*': unknown size
A negative subscript was used in an array, or there is an improper size designation.

C2134: '*identifier*': struct/union too large
The declared symbol is greater than 2^{32}. If the struct/union did not have a tag name, this message reads "<unnamed> struct/union too large."

C2135: missing ')' in macro expansion
A macro reference with arguments is missing a closing parenthesis.

C2137: empty character constant
The single quotes delimiting a character constant must contain one character. For example, the declaration char a = ' ' is illegal. To represent a null character constant, use an escape sequence, such as '\0'.

C2138: unmatched close comment '*/'

The compiler detected */ without a matching /*. This usually indicates an attempt to use nested comments, which is illegal.

C2139: type following *type* is illegal

There is an illegal type combination, such as the following:

```
long char a;   /* Illegal */
```

C2140: argument type cannot be function returning...

A function is declared as a formal parameter of another function, as in the following example:

```
int funcl(a)
   int a( );   /* Illegal */
```

C2141: value out of range for enum constant

An enumerated constant has a value outside the range of values allowed for type **int**.

C2142: ellipsis requires three periods

The compiler has detected the token ".." and assumes "..." was intended.

C2143: syntax error: missing '*token*' before '*token*'

The compiler has detected a syntax error which it thinks is a missing token prior to the specified token. The compiler inserts the first token and attempts to continue parsing. Note that even if the compiler has guessed correctly and inserted the correct token, the compile fails until the user makes the change to the source file .

C2144: syntax error: missing '*token*' before type '*type*'

Same as C2143, except that the second token is known to be a type, such as **int** or **float**.

C2145: syntax error: missing '*token*' before identifier

Same as C2143, except that the second token is an identifier whose name is not currently known. This can happen in certain situations involving look-ahead tokens.

C2146: syntax error: missing 'token' before identifier '*identifier*'

Same as C2145, except that the identifier is listed.

C2147: array: unknown size

An operation has been done on an unsized array which requires knowledge of the array size, e.g.:

```
struct foo *p;
...
p[2];
```

where **struct foo** has not been defined at the time the **p[2]** is seen. You may also get this message when attempting to do arithmetic with a pointer to **void.** To correct, cast the pointer to an object of known size.

C2148: array too large
You used an array larger than 2^{32} bytes.

C2149: *'identifier'*: **named bit-field cannot have zero width**
Bit-fields of zero width must be unnamed.

C2150:*'identifier'*: **bit-field must have type int, signed int, or unsigned int**
You used compile option -Za to force ANSI conformance, bu declared a bit-field with a type other than those permitted.

C2151: more than one cdecl/fortran/pascal attribute specified
You gave more than one of the keywords cdecl, fortran, or pascal in a declaration.

C2152:*'operator'* **: pointers to a function with different attributes**
The function pointer operands of the specified *'operator'* have differing near or far attributes or different language (**cdecl** or **fortran/pascal**) attributes.

```
int far foo ();          /* far function */

int (near *fp) () = foo();     /* near func ptr - ERROR */
```

Or:

```
int pascal foo(int, int);  /* pascal function */

int (*fp) () = foo;/* C function pointer - ERROR */
```

C2153: hex constants must have at least 1 hex digit
You used the form \x, which is not valid syntax for a hexadecimal constant.

C2159: more than one storage class specified
You declared a variable with more than one storage class specifier, such as "extern static f;."

C2172: '*functionname***': actual is not a pointer: parameter** *n*
This message is generated when the *n*th parameter (of the *m*th parameter list) of *functionname* is a structure or a union and the corresponding formal parameter is a pointer to **void**.

```
int function(void *)
struct bar
    {
     int i,j,k;
     } *foo;
main()
{
     function(*foo);     /*  illegal to pass a structure by value */
                              /*  to a pointer to void                */

}
```

C2173: '*functionname*': actual is not a pointer: parameter *n*, parameter list *m*

This message is generated when the *n*th parameter (of the *m*th parameter list) of *functionname* is a structure or a union and the corresponding formal parameter is a pointer to **void**. (See example in C2172.)

C2177: constant too big

Information is lost because a constant value is too large to be represented in the type to which it is assigned.

Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compiling and linking. The number in square brackets [ ] at the end of each message gives the minimum warning level that must be set for the message to appear.

C4001: macro '*identifier*': requires parameters [1]
The given *identifier* was defined as a macro taking one or more arguments, but the *identifier* is used in the program without arguments.

C4002: too many actual parameters for macro '*identifier*' [1]
The number of arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

C4003: not enough actual parameters for macro '*identifier*' [1]
The number of arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

C4004: missing close parenthesis after 'defined' [1]
The closing parenthesis is missing from an **#if defined** phrase.

C4005: '*identifier*' : redefinition [1]
The given *identifier* is redefined.

C4006: #undef expected an identifier [1]
The name of the identifier whose definition is to be removed must be given with the **#undef** directive.

C4009: string too big, trailing chars truncated [1]
A string exceeds the compiler limit on string size. To correct this problem, you must break the string down into two or more strings.

C4011: identifier truncated to '*identifier*' [1]
Only the first 31 characters of an identifier are significant.

C4014: '*identifier*' : bit-field type must be unsigned [1]
Bit fields must be declared as **unsigned** integer types. A conversion has been supplied.

C4015: '*identifier*' : bit-field type must be integral [1]
Bit fields must be declared as **unsigned** integral types. A conversion has been supplied.

C4016: '*name*': no function return type [2]
No function declaration or definition for *name* has been given. The default return type of **int** is assumed.

C4017: cast of int expression to far pointer [1]
A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer produces an address with a meaningless segment value.

C4020: '*name*' too many actual parameters [1]
The number of arguments specified in a call to function *name* is greater than the number of parameters specified in the argument type list or in the function definition.

C4021: '*name*': too few actual parameters [1]
The number of arguments specified in a call to function *name* is less than the number of parameters specified in the argument type list or in the function definition.

C4022: '*name*': pointer mismatch : parameter *n* [1]
The given parameter has a different pointer type than is specified in the argument type list or the function definition for the named function.

C4024: '*name*': different types : parameter *n* [1]
The type of the given parameter in a function call does not agree with the argument type list or the function definition for the named function.

C4025: function declaration specified variable argument list [1]
The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.

C4026: function was declared with formal argument list [1]
The function was declared to take arguments, but the function definition does not declare formal parameters.

C4027: function was declared without formal argument list [1]
The argument type list consists of the word **void**. The function was declared to take no argument, but formal parameters are declared in the function definition, or arguments are given in a call to the function.

C4028: parameter *n* declaration different [1]
The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.

C4029: declared parameter list different from definition [1]
The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.

C4030: first parameter list is longer than the second [1]
A function is declared more than once, and the argument type lists in the declarations differ.

C4031: second parameter list is longer than the first [1]
A function is declared more than once, and the argument type lists in the declarations differ.

C4032: unnamed struct/union as parameter [1]
The structure or union type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.

C4033: function must return a value [2]
A function is expected to return a value unless it is declared as **void**.

C4034: sizeof returns 0 [1]
The **sizeof** operator is applied to an operand that yields a size of zero.

C4035: '*function*' :no return value [2]
A function declared to return a value does not do so.

C4036: unexpected formal parameter list [1]
A formal parameter list is given in a function declaration and is ignored.

C4037: '*identifier*' : formal parameters ignored [1]
Formal parameters appeared in a function declaration, for example:

```
extern int *f(a,b,c);
```

The formal parameters are ignored.

C4038: :'*identifier*' : formal parameter has bad storage class [1]
Formal parameters must have **auto** or **register** storage class.

C4039: '*identifier*' : function used as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

C4040: near/far/huge on '*identifier*' ignored [1]
The **near**, **far**, or **huge** keyword has no effect in the declaration of the given '*identifier*' and is ignored.

C4041: formal parameter on *'identifier'* is redefined [1].
The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function.

C4042: '*identifier*' : has bad storage class [1]
The specified storage class cannot be used in this context. For example, function parameters cannot be given **extern** class. The default storage class for that context is used in place of the illegal class.

C4043: '*identifier*' : void type changed to int [1]
You can declare only functions as having the **void** type.

C4044: huge on '*identifier*' ignored, must be an array [1]
The **huge** keyword can only be used in array declarations.

C4045: '*identifier*' : array bounds overflow [1]
Too many initializers are present for the given array. The excess initializers are ignored.

C4046: '&' on function/array, ignored [1]
You cannot apply the address-of operator **&** to a function or array identifier.

C4047: '*operator*' : different levels of indirection [1]
An expression involving the specified operator has inconsistent levels of indirection. For example:

```
char **p;       /* Two levels of indirection */
char *q;        /* One level of indirection */
  •
  •
  •
p=q;            /* Different levels of indirection */
```

C4048: array's declared subscripts differ [1]
An array is declared twice with differing sizes. The larger size is used.

C4049: '*operator*' : indirection to different types [1]
The indirection operator * is used in an expression to get access to values of different types.

C4051: data conversion [3]
Two data items in an expression had different types, causing the type of one item to be converted.

C4052: different enum types [1]
Two different **enum** types are used in an expression.

C4053: at least one void operand [1]
An expression with type **void** is used as an operand.

C4056: overflow in constant arithmetic [1]
The result of an operation exceeds 0x7FFFFFFF.

C4057: overflow in constant multiplication [1]
The result of an operation exceeds 0x7FFFFFFF.

C4058: address of frame variable taken, DS != SS [1]
Program was compiled with the default data segment (DS) not equal to the stack segment (SS), and you tried to point to a frame variable with a **near** pointer

C4059: segment lost in conversion [1]
The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.

C4060: conversion of long address to short address [1]
The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) results in the loss of the segment address.

C4061: long/short mismatch in argument: conversion supplied [1]
An integral type is assigned to an integer of a different size, causing a conversion to take place. For example, a **long** is given where a **short** was declared.

C4062: near/far mismatch in argument: conversion supplied [1]
A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.

C4063: '*identifier*' : function too large for post-optimizer [0]
The named function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

C4064: procedure too large, skipping [loop inversion or branch sequence or cross jump] optimization and continuing [0]
Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

C4065: recoverable heap overflow in post optimizer - some optimizations may be missed [0]
Some optimizations are skipped because not enough space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

C4066: local symbol table overflow - some local symbols may be missing in listings [1]
The listing generator ran out of heap space for local variables, so source listing might not contain symbol-table information for all local variables.

C4067: unexpected characters following '*identifier*' directive - newline expected [1]
There are extra characters following a preprocessor directive, such as the following :

```
#endif              NO_EXT_KEYS
```

This is accepted in in the IBM C Compiler Version 1.00 but not in IBM C/2. IBM C/2 requires comment delimiters, such as the following:

```
#endif              /* NO_EXT_KEYS */
```

C4068: unknown pragma [1]
The compiler does not recognize the pragma you used and ignores this pragma.

C4069: conversion of near pointer to long integer [1]
A **near** pointer is being converted to a long integer, which involves extending the high order word with the current data segment value.

C4071: '*identifier*': no function prototype given [3]
You did not supply an argument-type list for the given identifier.

C4072 : insufficient memory to process debugging information [1]
Your computer lacks enough memory to compile this program using the -Zi switch.

C4073: scoping too deep, deepest scoping merged when debugging [1]
The visibility control of identifiers in deeply-nested blocks exceeds a built-in limit. Variables in all the deepest levels will be visibile to CodeView during debugging.

C4074 : non standard extension used -'*description*' [3]
You used a valid construction which is not recognized by the proposed ANSI standard for C. The *description* string may be one of the following:

trailing ',' used for variable argument list
cast on lvalue
extended initializer form
benign typedef redefinition
redefined extern to static
macro formals in strings
missing ';' following last struct/union member
bitfield types other than int

C4075: size of switch expression or case constant too large - converted to int [1]
You used a switch expression that evaluated to more than 32767.

C4076: '*type*' : may be used on integral types only [1]
You used the given keyword with a non-integral data type.

C4077: unknown check_stack option [1]
You gave an incorrect argument to pragma check_stack.

C4078: missing ')' [1]
When using the (*arguments*) form of pragmas, you omitted the closing parenthesis.

C4084: expected a pragma keyword [1]
You used an unknown identifier in a pragma.

C4085: expected [on|off] [1]
The argument in the parenthesized form of the check_stack pragma must be either "on" or "off."

C4087: *'name'* **: declared with 'void' parameter list [1]**
A function declared with a **void** parameter list was called with actual arguments.

C4088: *'name'* **: pointer mismatch: parameter** *n*, **parameter list** *m* **[1]**
You called a function with an actual parameter of a different pointer type from the formal parameter.

C4089: *'name'* **: different types: parameter** *n*, **parameter list** *m* **[1]**
You called a function with an actual parameter of a different type from the formal parameters.

C4093: unescaped newline in character constant in non-active code
The preprocessor found an unmatched single quote or double quote on a single line within an **#if/#ifdef/#elif/#else** block which is being skipped because of a false entry condition.

Command Line Messages

The following messages indicate errors on the command line that you use to call the compiler. If possible, the compiler continues operation, printing a warning message. In some cases, command line errors are fatal and the compiler stops processing.

Fatal Error Messages

D1000:UNKNOWN COMMAND LINE FATAL ERROR
An unforeseen error condition has been detected by the compiler. Please report this error to your authorized IBM dealer.

D1001: could not execute *'pass'*
The specified compiler file could not be found, or there is not enough space in storage.

D1002: too many open files, cannot redirect *'filename'*
No more file handles are available to redirect the output of the **-P** option to a file. Try editing the CONFIG.SYS file and increasing the value *num* on the line **files**=*num* (if *num* is less than 20.)

Error Messages

D2000:UNKNOWN COMMAND LINE ERROR
An unforeseen error condition has been detected by the compiler. Please report this error to your authorized IBM dealer.

D2001: too many symbols predefined with –D
The limit on command line definitions is normally 16; the **/U** option can increase the limit to 20.

D2002: a previously-defined model specification has been overridden.
Two different storage models are specified; the model specified last is used.

D2003: missing source file name
You must give the name of the source file to be compiled.

D2004: too many commas
Too many commas appear on the command line.

D2005: comma needed before :*'filename'*
The fields in the command line must be set off by commas.

D2006: a file name (not a path name) is required
The name of a directory is given where the name of a file is required.

D2008: too many *option* **flags in** *'string'*
Too many letters are given with a specific option (for example, with the **/O** option).

D2009: unknown option *'c'* **in** *'option'*
One of the letters in the given option is not recognized.

D2010: unknown floating-point option
The specified floating-point option (an **/FP** option) is not one of the five valid options.

D2011: only one floating-point model allowed
You can only give one of the five floating-point (**/FP**) options on the command line.

D2012: too many linker flags on command line
For compile-and-link (**CL**) only, you attempted to pass more than 128 separate options and object files to the linker.

D2013: incomplete model specification
The A*string* option requires all three character (data-pointer size, code-pointer size, and segment setup) in *string*.

D2014: -ND not allowed with -Ad
You cannot rename the default data segment unless you give the **-Au** option (if the stack segment is not equal to the data segment, load the data segment).

D2015: assembly files are not handled
You specified a filename with the extension .ASM. The compiler cannot invoke MASM automatically, so it cannot assemble these files.

D2016: -Gw and -ND *name* **are incompatible**
You cannot rename the default data segment to *name* when you give the **-G2** option because **-Gw** also requires **-Aw**.

D2017: -Gw and -Au flags are incompatible
You cannot use the **-Au** option (if the stack segment does not equal the data segment, load the data segment) with **-Gw** because **-Gw** also requires **-Aw**.

D2018: cannot open linker cmd file

The compiler cannot open the response file used to pass object-file names and options to the linker. One possible cause of this error is the existence of another file that is a read-only file with the same name as the response file.

D2019: cannot overwrite the source file, *'filename'*

The source file specified an output file name. The compiler does not allow the source file to be overwritten by one of the compiler output files.

D2020: -Gc option requires extended keywords to be enabled (-Ze)

The **-Gc** option requires the extended keyword **cdecl** to be enabled if the library functions are to be accessible.

D2021: invalid numerical argument *'string'*

You specified a non-numerical string following an option that requires a numerical argument.

D2022: cannot open help file *'filename'*

The driver expects the help file to be in the same directory it is in, or in the path.

D2027: cannot link file *'filename'*

You specified a filename with the extension .OBJ. This is not a valid extension as a source file name for the **CC** command.

Warning Messages

D4000:UNKNOWN COMMAND LINE WARNING

An unforeseen error condition has been detected by the compiler. Please report this error to your authorized IBM dealer.

D4001: listing has precedence over assembly output

Two different listing options were chosen; the assembly listing is not created.

D4002: ignoring unknown flag *'string'*

One of the options given on the command line is not recognized and is ignored.

D4003: 80186/286 selected over 8086 for code generation

Both **/G1** and **/G2** are selected.

D4004: optimizing for time over space

This message confirms that the **/Ot** option is used for optimizing.

D4005: could not execute '*name*',
please insert diskette and press any key.
One of the compiler passes cannot be found on the current disk. Insert the disk containing the named file and press any key. When the current subdirectory is on a fixed disk, the job must be halted. Move the necessary file to the subdirectory and recompile.

D4006: only one of -P/-E/-EP allowed, -P selected
Only one preprocessor option can be specified at one time.

D4007: -C ignored (must also specify -P or -E or -EP)
The -C option must be used with one of the preprocessor output flags, -E, -EP, or -P.

D4009: threshold only for far/huge data, ignored
The **-Gt** option cannot be used in memory models that have near data pointers. The **-Gt** option can be used only with compact-, large-, and huge-memory models.

D4010: -Gp not implemented, ignored
The DOS version of the compiler does not allow profiling.

D4011: preprocessing overrides source listing
The compiler produces only a preprocessor listing because it cannot produce both a source listing and a preprocessor listing at the same time.

D4012: function declarations override source listing
The compiler cannot produce both a source-listing file and the function prototype declarations at the same time.

D4013: combined listing has precedence over object listing
When **-Fc** is specified along with either **-Fl** or **-Fa**, the combined listing (**-Fc**) is created.

D4014: invalid value *n* for '*identifier*'. Default *m* is used
You used an incorrect numerical value for the given switch.

Compiler Limits

To operate IBM C/2, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

The following table summarizes the limits imposed by C/2. If your program exceeds one of these limits, an error message informs you of the problem.

Program Item	Description	Limit
String Literals	Maximum length of a string, including the ending null character (\0).	512 bytes
Constants	Maximum size of a constant. The type determines the maximum size of a constant. See the *IBM C/2 Fundamentals* book for a discussion of constants.	
Identifiers	Maximum length of an identifier.	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions.	10 levels
Preprocessor Directives	Maximum size of a macro definition for a macro with no arguments.	512 bytes
	Maximum length of a macro argument.	512 bytes
	Maximum level of nesting for **#if**, **#ifdef**, and **#ifndef** directives.	32 levels
	Maximum level of nesting for include files.	9 levels
Input Files	Maximum number of files processed by CL driver (includes .C and .OBJ files)	128 files

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler finds a function or program that is too large or too complex to be processed, it produces an error message to that effect.

Linker Error Messages

This section lists error messages produced by the IBM Linker.

Fatal errors cause the linker to stop running. Fatal error messages have the following format:

location: **fatal error L1***xxx*: *message text*

Non-fatal errors indicate problems in the executable file. LINK produces the executable file (and sets the error bit in the header if for protected mode). Non-fatal error messages have the following format:

location: **error L2** *xxx*: *message text*

Warnings indicate possible problems in the executable file. LINK produces the executable file (it does not set the error bit in the header if for protected mode). Warnings have the following format:

location: **error L4***xxx*: *message text*

In these messages, *location* is the input file associated with the error, or LINK if there is not input file. If the input file is a module definitions file, the line number will be included, as shown below:

foo.def(3): fatal error L1030:
missing internal name

If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ

The following error messages may appear when you link object files with LINK.

L1001 *option* **: option name ambiguous**
A unique option name does not appear after the option indicator (/). For example, the command

LINK /N main;

produces this error, since LINK cannot tell which of the three options beginning with the letter **N** is intended.

L1002 *option* **: unrecognized option name**
An unrecognized character followed the option indicator (/), as in the following example:

LINK /ABCDEF main;

L1003 *option* **: MAP symbol limit too high**
The specified symbol limit value following the MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.

L1004 *option* **: invalid numeric value**
An incorrect value appeared for one of the linker options. For example, a character string is entered for an option that requires a numeric value.

L1005 *option* **: packing limit exceeds 65536 bytes**
The number following the /PACKCODE option is greater than 65536.

L1006 *option* **: stack size exceeds 65534 bytes**
The size you specified for the stack in the /STACK option of the LINK command is more than 65534 bytes.

L1007 *option* **: interrupt number exceeds 255**
You gave a number greater than 255 as a value for the /OVERLAYINTERRUPT option.

L1008 *option* **: segment limit set too high**
The specified limit on the /SEGMENTS option is greater than 1024 using the /SEGMENTS

L1009 *option* **: CPARMAXALLOC : illegal value**
The number you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.

L1020 no object modules specified
You did not specify any object-file names to the linker.

L1021 cannot nest response files
A response file occurs within a response file.

L1022 response line too long
A line in a response file is longer than 127 characters.

L1023 terminated by user
You entered **Ctrl** + **C**.

L1024 nested right parentheses
You typed the contents of an overlay incorrectly on the command line.

L1025 nested left parentheses
You typed the contents of an overlay incorrectly on the command line.

L1026 unmatched right parenthesis
A right parenthesis is missing from the contents specification of an overlay on the command line.

L1027 unmatched left parenthesis
A left parenthesis is missing from the contents specification of an overlay on the command line.

L1030 missing internal name
In the module definitions file, when you specify an import by entry number, you must give an internal name, so the linker can identify references to the import.

L1031 module description redefined
In the module definitions file, a module description specified with the DESCRIPTION keyword is given more than once.

L1032 module name redefined
In the module definitions file, the module name is defined more than once with the NAME or LIBRARY keyword.

L1040 too many exported entries
An attempt is made to export more than 3072 names.

L1041 resident-name table overflow
The total length of all resident names, plus three bytes per name, is greater than 65534.

L1042 nonresident-name table overflow

The total length of all nonresident names, plus three bytes per name, is greater than 65534.

L1043 relocation table overflow

There are more than 65536 load-time relocations for a single segment.

L1044 imported-name table overflow

The total length of all the imported names, plus one byte per name, is greater than 65534 bytes.

L1045 too many TYPDEF records

An object module contains more than 255 TYPDEF records. These records describe communal variables. This error can only appear with programs produced by compilers that support communal variables.

L1046 too many external symbols in one module

An object module specifies more than the limit of 1023 external symbols. Break the module into smaller parts.

L1047 too many group, segment, and class names in one module

The program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

L1048 too many segments in one module

An object module has more than 255 segments. Split the module or combine segments.

L1049 too many segments

The program has more than the maximum number of segments. The SEGMENTS option specifies the maximum allowed number; the default is 128. Relink using the /SEGMENTS option with an appropriate number of segments.

L1050 too many groups in one module

The linker found more than 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module.

L1051 too many groups

The program defines more than 20 groups, not counting DGROUP. Reduce the number of groups.

L1052 too many libraries

An attempt is made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.

L1053 symbol table overflow

The program has more than 256K bytes of symbolic information, such as public, external, segment, group, class, and file names). Combine modules or segments and recreate the object files. Eliminate a many public symbols as possible.

L1054 requested segment limit too high

The linker does not have enough memory to allocate tables describing the number of segments requested (the default is 128 or the value specified with the /SEGMENTS option).

Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 too many overlays

The program defines more than 63 overlays.

L1057 data record too large

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator (compiler or assembler) error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM dealer.

L1070 segment size exceeds 64K

A single segment contains more than 64K bytes of code or data. Try compiling, or assembling, and linking using the large model.

L1071 segment _TEXT larger than 65520 bytes

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this is increased to 16 for alignment purposes.

L1072 common area longer than 65536 bytes

The program has more than 64K bytes of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler/2. It occurs only with programs produced by IBM C/2 or other compilers that support communal variables.

L1073 file-segment limit exceeded

There are more than 255 physical or file segments.

L1074 *name* **: group larger than 64K bytes**

A group contained segments which total more than 65536 bytes.

L1075 entry table larger than 65535 bytes

Because of an excessive number of entry names, you have exceeded a linker table size limit. Reduce the number of names in the modules you are linking.

L1080 cannot open list file

The disk or the root directory is full. Delete or move files to make space.

L1081 out of space for run file

The disk on which .EXE file is being written is full.
Free more space on the disk and restart the linker.

L1082 stub .EXE file not found

The stub file specified in the module definitions file is not found.

L1083 cannot open run file

The disk or the root directory is full. Delete or move files to make space.

L1084 cannot create temporary file

The disk or root directory is full. Free more space in the directory and restart the linker.

L1085 cannot open temporary file

The disk or the root directory is full. Delete or move files to make space.

L1086 scratch file missing

Internal error. You should note the conditions when the error occurs and contact your authorized IBM Computer dealer.

L1087 unexpected end-of-file on scratch file

The disk with the temporary linker-output file is removed.

L1088 out of space for list file

The disk on which the listing file is being written is full. Free more space on the disk and restart the linker.

L1089 *filename* **: cannot open response file**

The linker could not find the specified response file. This usually indicates a typing error.

L1090 cannot reopen list file

The original disk is not replaced at the prompt. Restart the linker.

L1091 unexpected end-of-file on library

The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.

L1092 cannot open module definitions file

The specified module definitions file cannot be opened.

L1100 stub .EXE file invalid

The stub file specified in the definitions file is not a valid .EXE file.

L1101 invalid object module

One of the object modules is non-valid.

If the error persists after recompiling, contact your authorized IBM dealer.

L1102 unexpected end-of-file

A non-valid format for a library was found.

L1103 attempt to access data outside segment bounds

A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM dealer.

L1104 *filename* **: not valid library**

The specified file is not a valid library file. This error causes the linker to stop running.

L1110 DOSALLOCHUGE failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer.

L1111 DOSREALLOCHUGE failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer.

L1112 DOSGETHUGESHIFT failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer.

L1113 unresolved COMDEF; internal error

You should note the conditions when the error occurs and contact your authorized IBM dealer.

L1114 file not suitable for /EXEPACK; relink without

For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image. Relink without the EXEPACK option.

L2000 imported entry point

A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.

L2001 fixup(s) without data

A FIXUP record occurred without a data record immediately preceding it. This is probably a compiler error. See the *IBM Disk Operating System Technical Reference* book for more information on FIXUP.

L2002 fixup overflow near *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*

The following conditions can cause this error:

- A group is larger than 64K bytes
- The program contains an intersegment short jump or intersegment short call
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link
- An EXTRN declaration in an assembler-language source file appeared inside the body of a segment.

For example:

```
code    SEGMENT public  'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public  'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and recreate the object file.

L2003 intersegment self-relative fixup

An intersegment self-relative fixup is not allowed.

L2004 LOBYTE-type fixup overflow

A LOBYTE fixup produced an address overflow.

L2005 fixup type unsupported

A fixup type occurred that is not supported by the linker. This is probably a compiler error. You should note the conditions when the error occurs and contact your authorized IBM dealer.

L2010 too many fixups in LIDATA record

There are more fixups applying to a LIDATA record than will fit in the linker's 1024-byte buffer.

The buffer is divided between the data in the LIDATA record and run-time relocation items, which are 8 bytes apiece, so the maximum varies form 0 to 128. This is probably a compiler error.

L2011 *name* : NEAR/HUGE conflict

Conflicting NEAR and HUGE attributes are given for a communal variable.

This error can occur only with programs produced by compilers that support communal variables.

L2012 *name* : array-element size mismatch

A far communal array is declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler/2. It occurs only with IBM C/2 and any other compiler that supports far communal arrays.

L2013 LIDATA record too large

A LIDATA record in an object module contains more than 512 bytes of data. Most likely, an assembly module contains a very complex structure definition or a series of deeply-nested DUP operators. For example, the following structure definition causes this error:

```
alpha   DB     10DUP(11 DUP(12 DUP(13 DUP(...))))
```

Simplify the structure definition and reassemble. (LIDATA is a DOS term).

L2020 no automatic data segment

No group named DGROUP is declared.

L2021 library instance data not supported in real mode

The library module is directed to have instance data. This works in protected mode only.

L2022 *name* **alias** *internalname*: **export undefined**

A name is directed to be exported but is not defined anywhere.

L2023 *name* **alias** *internalname*: **export imported**

An imported name is directed to be exported.

L2024 *name* **: symbol already defined**

One of the special overlay symbols required for overlay support is defined by an object.

L2025 *name* **: symbol defined more than once**

Remove the extra symbol definition from the object file.

L2026 multiple definitions for entry ordinal *number*

More than one entry point name is assigned to the same ordinal.

L2027 *name* **: ordinal too large for export**

You tried to export more than 3072 names.

L2028 automatic data segment plus heap exceeds 64K

The size of DGROUP near data plus requested heap size is greater than 64K.

L2029 unresolved externals

One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries.

A list of the unresolved external references appears after the message, as shown in the following example:

```
_exit in file(s)
 main.obj (main.c)
_fopen in files(s)
 fileio.obj(fileio.c) main.obj(main.c)
```

The name that comes before **in file(s)** is the unresolved external symbol. On the next line is a list of object modules which have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 starting address not code (use class 'CODE')

You specified a starting address to the linker which is a segment that is not a CODE segment. Reclassify the segment to CODE, or correct the starting point.

L4001 frame-relative fixup, frame ignored

A fixup occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fixup is meaningless in protected mode, so the target segment is assumed for the frame segment.

L4002 frame-relative absolute fixup

A fixup occurred with a frame segment different from the target segment where both frame and target segments were absolute. This fixup is processed using base-offset arithmetic, but the warning is issued because the fixup may not be valid in OS/2 mode.

L4010 invalid alignment specification

The number following the **/ALIGNMENT** option is not a power of 2, or is not in numerical form.

L4011 PACKCODE value exceeding 65500 unreliable

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4012 load-high disables EXEPACK

The options **/HIGH** and **/EXEPACK** are mutually exclusive.

L4013 invalid option for new-format executable file ignored

If an OS/2 mode program is being produced, then the options **/CPARMAXALLOC, /DSALLOCATE, /EXEPACK, /NOGROUPASSOCIATION**, and **/OVERLAYINTERRUPT** are meaingless, and the linker ignores them.

L4014 invalid option for old-format executable file ignored

If a DOS format program is produced, the options **/ALIGNMENT**, **/NOFARCALLTRANSLATION**, and **/PACKCODE** are meaningless, and the linker ignores them.

L4020 *name* **: code-segment size exceeds 65500**

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4021 no stack segment

The program does not contain a stack segment defined with STACK combine type. This message should not appear for modules compiled with IBM C/2, but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

L4022 *name1, name2* **: groups overlap**

Two groups are defined such that one starts in the middle of another. This may occur if you defined segments in a module definitions file or assembly file and did not correctly order the segments by class.

L4023 *exportname* **: export internal-name conflict**

An exported name, or its associated internal name, conflict with an already-defined public symbol.

L4024 *name* **: multiple definitions for export name**

The name *name* is exported more than once with different internal names. All internal names except the first are ignored.

L4025 *name* **: import internal-name conflict**

An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored. The conflict may come from a definition in either the module definition file or an object file.

L4026 *modulename* **: self-imported**

The module definitions file directed that a name be imported from the module being produced.

L4027 *name* **: multiple definitions for import internal-name**

An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.

L4028 *name* **: segment already defined**

A segment is defined more than once with the same name in the module definitions file. Segments must have unique names for the linker. All definitions with the same name after the first are ignored.

L4029 *name* **: DGROUP segment converted to type data**

A segment which is a member of DGROUP is defined as type CODE in a module definition file or object file.

This probably happened because a CLASS keyword in a SEGMENTS statement is not given.

L4030 *name* **: segment attributes changed to conform with automatic data segment**

The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute. For example, the *shared* attrbuté is NONSHARED and the *instance* is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.

L4031 *name* **: segment declared in more than one group**

A segment is declared to be a member of two different groups. Correct the source file and recreate the object files.

L4032 *name* **: code-group size exceeds 65500 bytes**

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4034 more than 239 overlay segments; extra put in root

You specified an overlay structure containing more than 239 segments. The extra segments have been assigned to the root overlay.

L4036 no automatic data segment

L4040 NON-CONFORMING : obsolete

In the module definitions file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.

L4041 HUGE segments not yet supported

This feature is not yet implemented in the linker.

L4042 cannot open old version

An old version of the EXE file, specified with the OLD keyword in the module definitions file, could not be opened.

L4043 old version not segmented-executable format

The old version of the .EXE file, specified with the OLD keyword in the module definitions file, does not conform to segmented-executable format.

L4045 : <name> : is name of output file

The user created a dynamic link library file without specifying an extension. In such cases, the linker supplies an extension of ".DLL." This is to warn the user in case he expected a ".EXE" file to be generated.

L4046 : module name different from output filename

The user specified a module name via the NAME or LIBRARY statement in the definitions file which is different from the output file (.EXE or .DLL) name. This will likely cause problems in BINDING the file or in using it in OS/2 mode so the user should rename the file to match the module name before it is executed.

L4050 too many public symbols

The **/MAP** option is used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (the default is 2048 symbols). The linker produces an unsorted listing of the public symbols. Relink using **/MAP**:*number*.

L4051 *filename* **: cannot find library**

The linker could not find the specified file. Enter a new filename, a new path specification, or both.

L4053 VM.TMP : illegal filename; ignored

VM.TMP appears as an object-file name. Rename the file and rerun the linker.

L4054 *filename* **: cannot find file**

The linker could not find the specified file. Enter a new filename, a new path specification, or both.

Linker Limits

The table below summarizes the limits imposed by the linker. If you find one of these limits, you may adjust your program so that the linker can accommodate it.

Item	Limit
Symbol table	256K
Load-time relocations (for DOS programs)	Default is 32K. If **/EXEPACK** is used, the maximum is 512K.
Public symbols	The range 7700-8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defined DGROUP so the effective maximum is 20.
Overlays	63

Item	Limit
Segments	128 by default; however, this maximum can be set as high as 1024 by using the **/SEGMENTS** option of the LINK command.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64K

Library Manager Error Messages

Error messages produced by the IBM Library Manager, LIB, have one of the following formats:

- *filename*|LIB : fatal erro r U1*xxx* : *messagetext*
- *filename*|LIB : warning U4*xxx* : *messagetext*

The message begins with the input filename (*filename*), if one exists, or with the name of the utility. LIB may display the following error messages:

U1150 page size too small
The page size of an input library is too small, which indicates a non-valid input .LIB file.

U1151 syntax error : illegal file specification
You gave a command operator, such as a minus sign (-), without a module name following it.

U1152 syntax error : option name missing
You gave a forward slash (/) without a value following it.

U1153 syntax error : option value missing
You gave the **/PAGESIZE** option without a value following it.

U1154 option unknown
An unknown option is given. Currently, LIB recognizes the **/PAGESIZE** option only.

U1155 syntax error : illegal input
The given command did not follow correct LIB syntax.

U1156 syntax error
The given command did not follow correct LIB syntax.

U1157 comma or new line missing
A comma or carriage return is expected in the command line, but did not appear. This may indicate an inappropriately placed comma, as in the following line:

LIB math.lib,-mod1 + mod2;

The line should have been entered as follows:

LIB math.lib -mod1 + mod2;

U1158 terminator missing

Either the response to the **Output library:** prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 cannot rename old library

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection. Change the protection of the old .BAK version.

U1162 cannot reopen library

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 error writing to cross-reference file

The disk or root directory is full. Delete or move files to make space.

U1170 too many symbols

More than 4609 symbols appeared in the library file.

U1171 insufficient memory

LIB did not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.

U1172 no more virtual memory

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1173 internal failure

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1174 mark : not allocated

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1175 free : not allocated

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1180 write to extract file failed

The disk or root directory is full. Delete or move files to make space.

U1181 write to library file failed

The disk or root directory is full. Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory is full, or the specified extract file already exists with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 cannot open response file

The response file was not found.

U1184 unexpected end-of-file on command input

An end-of-file character is received prematurely in response to a prompt.

U1185 cannot create new library

The disk or root directory is full, to the library file already exists with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 error writing to new library

The disk or root directory is full. Delete or move files to make space.

U1187 cannot open VM.TMP

The disk or root directory is full. Delete or move files to make space.

U1188 cannot write to VM

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1189 cannot read from VM

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1190 DOSALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1191 DOSREALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1192 DOSGETHUGESHIFT failed
You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1200 *name* **: invalid library header**
The input library file has a non-valid format. It is either not a library file, or it has been corrupted.

U1203 *name* **: invalid object module near** *location*
The module specified by *name* is not a valid object module.

U4150 *modulename* **: module redefinition ignored**
A module is specified to be added to a library, but a module with the same name is already in the library. Or, a module with the same name is found more than once in the library.

U4151 *symbol(modulename)* **: symbol redefinition ignored**
The specified symbol is defined in more than one module.

U4152 *filename* **: cannot create listing**
The directory or disk is full, or the cross-reference listing file already exists with read-only protection. Make space on the disk or change the protection of the cross-reference listing file.

U4153 *number* **: page size too small; ignored**
The value specified in the **/PAGESIZE** option is less than 16.

U4155 *modulename* **: module not in library; ignored**
The specified module is not found in the input library.

U4156 *libraryname* **: output-library specification ignored**
An output library is specified in addition to a new library name. For example, specifying

LIB new.lib + one.obj,new.lst,new.lib

where **new.lib** does not already exist causes this error.

U4157 *filename* **: cannot access file**
LIB is unable to open the specified file.

U4158 *libraryname* **: invalid library header; file ignored**
The input library has an incorrect format.

U4159 *filename* **: invalid format** *hexnumber*; **file ignored**
The signature byte or word, *hexnumber*, of an input file is not one of the recognized types.

MAKE Error Messages

Error messages displayed by the IBM Program Maintenance Utility, MAKE, have one of the following formats:

- *filename*|MAKE : fatal error U1*xxx* : *messagetext*
- *filename*|MAKE : warning U4*xxx* : *messagetext*

The message begins with the input filename (*filename*), if one exists, or with the name of the utility. MAKE produces the following error messages:

U1001 macro definition larger than *number*

A single macro is defined to have a value string longer than the number stated.

Try rewriting the MAKE description file to split the macro into two or more smaller ones.

U1002 infinitely recursive macro

A circular chain of macros is defined, as in the following example:

```
A = $(B)
B = $(C)
C = $(A)
```

U1003 out of memory

MAKE ran out of memory for processing the MAKE description file.

Try to reduce the size of the MAKE description file by reorganizing or splitting it.

U1004 syntax error : macro name missing

The MAKE description file contained a macro definition with no left side (that is, a line beginning with =)

U1005 syntax error : colon missing

A line that should be an outfile/infile line lacked a colon indicating the separation between outfile and infile. MAKE expects any line following a blank line to be an outfile/infile line.

U1006 *targetname* **: macro expansion larger than** *number*

A single macro expansion, plus the length of any string it may be concatenated to, is longer than the number stated. Try rewriting the MAKE description file to split the macro into two or more smaller ones.

U1007 multiple sources

An inference rule is defined more than once.

U1008 *name* **: cannot find file or directory**

The file or directory specified by *name* could not be found.

U1009 *command* **: argument list too long**

A command line in the MAKE description file is longer than 128 bytes, which is the maximum that DOS allows. Rewrite the commands to use shorter argument lists.

U1010 *filename* **: permission denied**

The file specified by *filename* is a read-only file.

U1011 *filename* **: not enough memory**

Not enough memory is available for MAKE to run a program.

U1012 *filename* **: unknown error**

You should note the conditions when the error occurs and contact your authorized IBM dealer.

U1013 *command* **: error** *errcode* **(ignored)**

One of the programs or commands called in the MAKE description file returned with a nonzero error code.

If MAKE is run with the **/I** option, MAKE displays **(ignored)** and continues. Otherwise, MAKE stops running.

U4000 *filename* **: target does not exist**

This usually does not indicate an error.

It warns you that the target file does not exist. MAKE runs any commands given in the block description since in many cases the outfile file will be created by a later command in the MAKE

U4001 dependent *filename* **does not exist; target** *filename* **not built**

MAKE could not continue because a required infile file did not exist. Make sure that all named files are present and that they are spelled correctly in the MAKE description file.

U4014 usage : make [/n] [/d] [/i] [/s] [name = value...] file

MAKE has not been called correctly. Try entering the command line again with the syntax shown in the message.

EXEMOD Error Messages

Error messages from the IBM EXE File Header Utility, EXEMOD, have one of the following formats:

- *filename*|EXEMOD : fatal error U1*xxx* : *messagetext*
- *filename*|EXEMOD : warning U4*xxx* : *messagetext*

The message begins with the input filename (*filename*), if one exists, or with the name of the utility. EXEMOD produces the following error messages:

U1050 usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]
You did not specify the EXEMOD command line properly. Try again using the syntax shown. Note that the option indicator can be either a slash (/) or a dash (-). The single brackets [] in the error message show your optional choice.

U1051 invalid .EXE file : bad header
The specified input file is not an executable file or has an incorrect file header.

U1052 invalid .EXE file : actual length less than reported
The second and third fields in the input-file header indicate a file size greater than the actual size.

U1053 cannot change load-high program
When the minimum allocation value and the maximum allocation value are both zero, you cannot change the file.

U1054 file not .EXE
EXEMOD adds the .EXE extension to any filename without an extension. In this case, no file with the given name and an .EXE extension was found.

U1055 *filename* **: cannot find file**
The file specified by *filename* was not found.

U1056 *filename* **: permission denied**
The file specified by *filename* is a read-only file.

U4050 packed file
The given file is a packed file. This is a warning only.

U4051 minimum allocation less than stack; correcting minimum

If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the changed request),

the minimum allocation value is adjusted. This is a warning message only; the change is still performed.

U4052 minimum allocation greater than maximum; correcting maximum

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted.

This is a warning message only; the change is still performed.

Errno Value Error Messages

This section lists and describes the values to which the *errno* variable can be set when an error occurs in a call to a library routine. Note that only some routines set the *errno* variable. See Chapter 5, "Library Routines" in this book for the routines that set *errno*.

An error message is associated with each *errno* value. This message, along with a message that you supply, can be printed by using the **perror** function.

The value of *errno* reflects the error value for the last call that set *errno*. The *errno* value is not automatically cleared by later successful calls. Thus, you should test for errors and print error messages immediately after a call to obtain accurate results.

The **errno.h** include file contains the definitions of the *errno* values. However, not all of the definitions given in **errno.h** are used under DOS. This section lists only the *errno* values used under DOS. For the complete listing of values, see the **errno.h** include file.

Also listed on this section are the errors produced by math routines when an error occurs. These errors correspond to the exception types defined in the **math.h** include file and returned by the **matherr** function when a math error occurs.

Errno Values

The following list gives the *errno* values used under DOS, the system error message corresponding to each value, and a brief description of the circumstances that caused the error.

Value	Message and Description
E2BIG	**Arg list too long.**
	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K bytes.
EACCESS	**Permission denied.**
	The permission setting of the file does not allow the specified access. This error can occur in a variety of circumstances. It signifies that an attempt was made to get access to a file (or, in some cases, a directory) in a way that is incompatible with the attributes of the file.
	For example, this error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under DOS 3.00 and later and OS/2, **EACCESS** can also indicate a locking or sharing violation.
	This error can also occur in an attempt to rename a file or directory or to remove an existing directory.
EBADF	**Bad file number.**
	The specified file handle is not a valid file handle value, does not refer to an open file, or an attempt was made to write to a file or device opened for read access (or the reverse).
EDEADLOCK	**Resource deadlock would occur.**
	Locking violation: the file cannot be locked after 10 attempts.

EDOM **Math argument.**

The argument to a math function is not in the domain of the function.

EEXIST **File exists.**

The **O_CREAT** and **O_EXCL** flags are specified when opening a file, but the named file already exists.

EINVAL **non-valid argument.**

A non-valid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file.

EMFILE **Too many open files.**

No more file handles are available, so no more files can be opened.

ENOENT **No such file or directory.**

The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a pathname does not specify an existing directory. It can also occur in OS/2 mode if a filename exceeds 8 characters, or if the filename extension exceeds 3 characters.

ENOEXEC **Exec format error.**

An attempt is made to run a file that is not executable or that has a non-valid executable file format.

ENOMEM **Not enough core.**

Not enough storage is available. This message can occur when not enough storage is available to run a child process or when the allocation request in an **sbrk** or **getcwd** call cannot be satisfied.

ENOSPC **No space left on the device.**

No more space for writing is available on the device. (For example, the disk is full.)

ERANGE **Result too large.**

An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the pathname argument to the **getcwd** function is longer than expected).

EXDEV **Cross-device link.**

An attempt was made to move a file to a different device (using the **rename** function).

Math Errors

The following errors can be generated by the math routines of the C run-time library. These errors correspond to the exception types defined in **math.h** and returned by the **matherr** function when a math error occurs. See Chapter 4, "Include Files," for more information.

Error	Description
DOMAIN	An argument to the function is outside the domain of the function.
OVERFLOW	The result is too large to be represented in the return type of the function.
PLOSS	A partial loss of significance occurred.
SING	Argument singularity: an argument to the function has an illegal value (for example, passing the value zero to a function that requires a nonzero value).
TLOSS	A total loss of significance occurred.
UNDERFLOW	The result is too small to be represented.

CodeView Error Messages

CodeView displays an error message whenever it detects a command it cannot run. You might see any of the following error messages. Except for start-up errors, most errors stop the CodeView command in which the error occurred, but do not stop CodeView. For more information about CodeView, see the *IBM C/2 Compile, Link, and Run* book.

Bad address
You specified the address in an non-valid form. For example, you might have entered an address containing hexadecimal characters when the radix is decimal.

Bad breakpoint command
You typed an non-valid breakpoint number with the BREAKPOINT CLEAR, BREAKPOINT DISABLE, or BREAKPOINT ENABLE command. The number must be in the range of 0 through 19.

Bad flag
You specified an non-valid flag mnemonic with the REGISTER dialog command (**R**). Use one of the mnemonics that appears when you enter the command **RF**.

Bad format string
You specified a non-valid type specifier following an expression. Expressions used with the DISPLAY EXPRESSION, WATCH, WATCHPOINT, and TRACEPOINT commands can have **printf** type specifiers set off from the expression by a comma. The valid type specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some type specifiers can be preceded by the prefix **h** or **l**.

Bad radix (use 8, 10, or 16)
CodeView only uses octal, decimal, and hexadecimal radixes.

Bad register
You typed the REGISTER command (**R**) with an non-valid register name. Use **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, or **F**.

Bad type cast
The valid types for type-casting are the C types **void**, **char**, **int**, **short**, **long**, **signed**, **unsigned**, **float**, and **double**. The types **unsigned**, **signed**, **long**, and **short** can be combined with other

types (for example, **unsigned char**), as listed in the *IBM C/2 Fundamentals* book.

Bad type (use one of 'ABDILSTUW')
The valid dump types are ASCII (**A**), byte (**B**), integer (**I**), unsigned (**U**), word (**W**), doubleword (**E**), short real (**S**), long real (**L**), and ten-byte real (**T**).

Badly formed type
The type information in the symbol table of the file you are debugging is incorrect. If this message occurs, note the circumstances of the error and report it using the Reader's Comment Form.

Breakpoint "# or '" expected
You entered the BREAKPOINT CLEAR (**BC**), BREAKPOINT DISABLE (**BD**), or BREAKPOINT ENABLE (**BE**) commands with no argument. These commands require that you specify the number of the breakpoint at which CodeView is to act or that you specify an asterisk *, indicating that CodeView is to act on all breakpoints.

Cannot use struct or union as scalar
You cannot use a structure or union variable as a scalar value in a C expression. The address-of operator must precede structure or union variables, and a field specifier must follow them.

Can't find filename
CodeView cannot find the executable file you specified when you started. You probably misspelled the file name, or the file is in a different directory.

Constant too big
CodeView cannot accept a constant number larger than 4294967295 (0xFFFFFFFF).

Divide by zero
An expression in an argument of a dialog command attempts to divide by zero.

Expression too complex
An expression given as a dialog command argument is too complex. Simplify the expression.

Extra input ignored

You specified too many arguments to a command. CodeView evaluates the valid arguments and ignores the rest. Often in this situation, CodeView does not evaluate the arguments in the order that you intended.

Floating point error

If this message occurs, note the circumstances of the error and report it to your authorized IBM dealer.

Internal debugger error

If this message occurs, note the circumstances of the error and report it to your authorized IBM dealer.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

Missing ''

You specified a string as an argument to a dialog command, but you did not supply a closing double quote mark.

Missing ')'

You specified an argument to a dialog command as an expression containing a left parenthesis but no right parenthesis.

Missing '('

You specified an argument to a dialog command as an expression containing a right parenthesis but no left parenthesis.

Missing '['

You specified an argument to a dialog command as an expression containing a right bracket but no left bracket. This error can also occur if you specify a regular expression with a right bracket but no left bracket.

No closing single quote

You specified a character in an expression used as a dialog command argument, but the closing single quote is missing.

No code at this line number

You tried to set a breakpoint on a source line that does not correspond to code. The line might be a data declaration or a comment.

No match of regular expression

CodeView can find no match for the regular expression you specified with the SEARCH command or with the **Find** selection from the **Search** menu.

No previous regular expression

You selected **Previous** from the **Search** menu, but there was no previous match for the last regular expression specified.

No program to debug

You have run to the end of the program you are debugging. You must restart the program (using the RESTART command) before using any command that runs code.

No source lines at this address

The address you specified as an argument for the VIEW command (**V**) does not have any source lines. It might be an address in a library routine or an assembly-language module.

No such file/directory

A file you specified in a command argument or in response to a prompt does not exist. For example, this message appears when you select **Load** from the **File** menu and then enter the name of a nonexistent file.

No symbolic information

The program file you specified is not in the CodeView format. You cannot debug in source mode, but you can use assembly mode.

Not a text file

You attempted to load a file using the **Load** selection from the **File** menu or using the VIEW command, but the file is not a text file. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII range of 9 through 13 and 20 through 126.

Not an executable file

The file you specified for debugging when you started CodeView is not an executable file having the extension .EXE or .COM.

Not enough space

You typed the SHELL ESCAPE command (**!**) or selected **Shell** from the **File** menu, but there is not enough free storage to run COMMAND.COM . Because storage is released by code in the C start-up routines, this error always occurs if you try to use the

the code run commands (TRACE, PROGRAM STEP, or GO) to run the C start-up code, then try the SHELL ESCAPE command again. The message also occurs with assembly-language programs that do not specifically release storage. This message also appears when CodeView does not have enough memory space to load your program.

Object too big

You entered a TRACEPOINT command with a data object, such as an array, that is larger than 128 bytes. You can watch data objects larger than 128 bytes using the storage version of the TRACEPOINT command.

Operand types incorrect for this operation

An operand in a C expression had a type that is incompatible with the operation applied to it. For example, if you declare **p** as **char ***, then **? p*p** produces this error because a pointer cannot be multiplied by a pointer.

Operator must have a struct/union type

You used the one of the member selection operators **(->** or **.)** in an expression that does not refer to an element of a structure or a union.

Operator needs lvalue

You specified an expression that does not evaluate to an lvalue in an operation that requires an lvalue. For example, **? 3 = 100** is non-valid. See the *IBM C/2 Fundamentals* book for more information on lvalues.

Program terminated normally (*number*)

You ran your program to the end. The number displayed in parentheses is the exit code that your program returns to DOS or OS/2. You must use the RESTART command (or the **Start** menu selection) to start the program before running more code.

Register variable out of scope

You tried to specify a register variable using the period (.) operator and a function name. For example, if you are in a third-level function, you can display the value of a local variable called **local** in a second-level function called **parent** with the following command:

```
? parent.local
```

However, this command does not work if you declare **local** as a register variable.

Regular expression too complex
The regular expression you specified is too complex for CodeView to evaluate.

Regular expression too long
The regular expression you specified is too long for CodeView to evaluate.

Syntax error
You specified an non-valid command line for a dialog command. Check for an non-valid command letter. This message also appears if you enter an non-valid assembly-language instruction using the ASSEMBLE command. The error follows a caret that points to the first character that CodeView cannot interpret.

Too many breakpoints
You tried to specify a 21st breakpoint. Codeview permits only 20 breakpoints.

Too many open files
You do not have enough file handles for CodeView to operate correctly. You must specify more files in your CONFIG.SYS file. See your Disk Operating System book for information about using the CONFIG.SYS file.

Type conversion too complex
You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type is type casting to a structure or union type. An example of two levels of indirection is **char ****.

Unable to open file
CodeView cannot open a file that you specified in a command argument or in response to a prompt. For example, this message appears when you select **Load** from the **File** menu and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

Unknown symbol
You specified an identifier that is not in CodeView's symbol table. Check for a misspelling. CodeView cannot recognize a symbol name spelled with letters of the wrong case unless you turn off the **Case Sense** selection on the **Options** menu. Another potential cause for this message is if you try to use a local variable in an argument when you are not in the function in which you define the variable.

Unrecognized option *option* **- The valid options are /B, /C***command***, /F, /S, /T, OR /W**

You entered an non-valid option when starting CodeView. Retype the command line.

Usage: cv [options] file [arguments]

You failed to specify an executable file when you started CodeView. Try again with the syntax shown in the message.

Video mode changed without /S option

The program changed video modes from or to one of the graphics modes when screen swapping was not specified. You must use the /S option to specify screen swapping when you are debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program might be damaged.

Warning: packed file

You started CodeView with a packed file as the executable file. You can attempt to debug the program in assembly mode, but the packing routines at the start of the program might make this difficult. You cannot debug in source mode because EXEPACK strips all symbolic information from a file when it packs the file. This occurs with the /EXEPACK linker option.

Appendix B. Reentrant Functions

With OS/2, you can call operating system functions from within C programs. The facility of creating multiple threads of execution is illustrated in Appendix A of the *IBM C/2 Compile, Link, and Run* book. That appendix warns of the usage of non-reentrant functions with multiple threads.

A list of the reentrant library functions in the C run-time library for OS/2 is given here for reference.

abs	lfind	strchr	strupr
atol	longjmp	strcmp	swab
atoi	memccpy	strcmpi	tolower
bsearch	memchr	strcpy	toupper
chdir	memcmp	stricmp	utime
getch	memcpy	strlen	
getche	memicmp	strlwr	
getpid	memset	strncal	
halloc	movedata	strnicmp	
hfree	outp	strncpy	
inp	putch	strnset	
iota	qsort	strrchr	
kbhit	segread	strrev	
labs	setjmp	strset	
lsearch	strcat	strstr	

Appendix C. ASCII Characters

000	001	002	003	004	005	006	007	008	009
NUL	☺ SOH	☻ STX	♥ ETX	♦	♣	♠	BEL	◘ BS	HT
010	011	012	013	014	015	016	017	018	019
LF	VT	FF	CR	♫ SO	☼ SI	► DLE	◄ DC1	↕ DC2	!! DC3
020	021	022	023	024	025	026	027	028	029
¶ DC4	§ NAK	▬ SYN	↨ ETB	↑ CAN	↓ EM	→ SUB	← ESC	FS	GS
030	031	032	033	034	035	036	037	038	039
RS	US	SP	!	"	#	$	%	&	'
040	041	042	043	044	045	046	047	048	049
(	)	*	+	,	—	.	/	0	1
050	051	052	053	054	055	056	057	058	059
2	3	4	5	6	7	8	9	:	;
060	061	062	063	064	065	066	067	068	069
<	=	>	?	◌	A	B	C	D	E
070	071	072	073	074	075	076	077	078	079
F	G	H	I	J	K	L	M	N	O
080	081	082	083	084	085	086	087	088	089
P	Q	R	S	T	U	V	W	X	Y
090	091	092	093	094	095	096	097	098	099
Z	[	\	]	^	_	`	a	b	c
100	101	102	103	104	105	106	107	108	109
d	e	f	g	h	i	j	k	l	m
110	111	112	113	114	115	116	117	118	119
n	o	p	q	r	s	t	u	v	w
120	121	122	123	124	125	126	127	128	129
x	y	z	{	¦	}	~	⌂	Ç	ü

130	131	132	133	134	135	136	137	138	139
é	â	ä	à	å	ç BEL	ê	ë HT	è LF	ï VT
140	141	142	143	144	145	146	147	148	149
î FF	ì CR	Ä SO	Â SI	É	æ	Æ DC2	ô	ö DC4	ò
150	151	152	153	154	155	156	157	158	159
û	ù	ÿ CAN	ö	ü	¢ ESC	£	¥	Pt	ƒ
160	161	162	163	164	165	166	167	168	169
á	í	ó	ú	ñ	Ñ	a	o	¿	⌐
170	171	172	173	174	175	176	177	178	179
¬	½	¼	¡	«	»	▒	▒	▒	|
180	181	182	183	184	185	186	187	188	189
┤	╡	╢	╖	╕	╣	║	╗	╝	╜
190	191	192	193	194	195	196	197	198	199
╛	┐	└	┴	┬	├	─	┼	╞	╟
200	201	202	203	204	205	206	207	208	209
╚	╔	╩	╦	╠	═	╬	╧	╨	╤
210	211	212	213	214	215	216	217	218	219
╥	╙	╘	╒	╓	╫	╪	┘	┌	▉
220	221	222	223	224	225	226	227	228	229
▄	▌	▐	▀	α	β	Γ	∏	Σ	σ
230	231	232	233	234	235	236	237	238	239
µ	τ	Φ	Θ	Ω	δ	∞	Ø	∈	∩
240	241	242	243	244	245	246	247	248	249
≡	±	≥	≤	⌠	⌡	÷	≈	°	•
250	251	252	253	254	255				
·	√	∩	²	■	SP				

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Notes:

Notes:

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

84X1794