## Contents

### Part 1 1

#### 1 Introduction 1

1.1 About the C Library 3  
1.2 About This Manual 4  
1.3 Notational Conventions 6  

#### 2 Using C Library Routines 9

2.1 Introduction 11  
2.2 Identifying Functions and Macros 11  
2.3 Including Files 13  
2.4 Declaring Functions 14  
2.5 Stack Checking on Entry 15  
2.6 Argument Type Checking 16  
2.7 Error Handling 18  
2.8 File Names and Path Names 19  
2.9 Binary and Text Modes 22  
2.10 DOS Considerations 23  
2.11 Floating-Point Support 25  
2.12 Using Huge Arrays with Library Functions 26  

#### 3 Global Variables and Standard Types 29

3.1 Introduction 31  
3.2 _amblksz 31  
3.3 daylight, timezone, tzname 32  
3.4 _doserrno,_errno, sys_errnolist, sys_nerr 33  
3.5 _fmode 34  
3.6 _osmajor, _osminor 34  
3.7 environ, _psp 35  
3.8 Standard Types 36
4 Run-Time Routines by Category

4.1 Introduction 41
4.2 Buffer Manipulation 41
4.3 Character Classification and Conversion 42
4.4 Data Conversion 44
4.5 Directory Control 45
4.6 File Handling 45
4.7 Input and Output 46
4.8 Math 58
4.9 Memory Allocation 60
4.10 System Calls and Interrupts 64
4.11 Process Control 66
4.12 Searching and Sorting 70
4.13 String Manipulation 70
4.14 Time 72
4.15 Variable-Length Argument Lists 73
4.16 Miscellaneous 73

5 Include Files

5.1 Introduction 79
5.2 assert.h 80
5.3 conio.h 80
5.4 ctype.h 80
5.5 direct.h 81
5.6 dos.h 81
5.7 errno.h 82
5.8 fcntl.h 83
5.9 float.h 83
5.10 io.h 83
5.11 limits.h 84
5.12 malloc.h 84
5.13 math.h 84
5.14 memory.h 85
5.15 process.h 85
5.16 search.h 86
5.17 setjmp.h 86
5.18 share.h 86
5.19 signal.h 87
5.20 stdarg.h 87
5.21 stddef.h 87
5.22 stdio.h 88
5.23 stdlib.h 89
5.24 string.h 90
5.25 sys\locking.h 90
5.26 sys\stat.h 90
5.27 sys\timeb.h 91
5.28 sys\types.h 91
5.29 sys\utime.h 91
5.30 time.h 91
5.31 varargs.h 92
5.32 v2tov3.h 92

Part 2 Reference 93

Appendixes 535

A Error Messages 535

A.1 Introduction 537
A.2 errno Values 537
A.3 Math Errors 540

B Common Libraries for DOS, XENIX, UNIX, and ANSI 541

B.1 Introduction 543
B.2 Run-Time Routines 543
B.3 Global Variables 547
B.4 Include Files 548
B.5 Differences Between Routines
   Common to DOS and XENIX 550
Tables

Table 4.1  Forms of the spawn and exec Routines  69
Table A.1  errno Values and Their Meanings   538
Table R.1  printf Type Characters   378
Table R.2  printf Flag Characters   379
Table R.3  How printf Precision Values Affect Type   381
Table 0.4  scanf Type Characters   417
Part 1

Introduction 1
Using C Library Routines 9
Global Variables and Standard Types 29
Run-Time Routines by Category 39
Include Files 77
Chapter 1
Introduction

1.1 About the C Library  3
1.2 About This Manual  4
1.3 Notational Conventions  6
1.1 About the C Library

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

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

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

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

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

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

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

For programmers interested in taking advantage of the specific features of DOS, the library includes DOS interface functions. These functions allow DOS system calls and interrupts to be invoked from a C program. The library also contains console input and output functions to allow efficient
reading and writing from the user's console.

To take advantage of the compiler's type-checking capabilities, the include files that accompany the run-time library have been expanded. In addition to the definitions and declarations required by library functions and macros, the include files now contain function declarations with argument-type lists. The argument-type lists enable type checking for calls to library functions. This feature can be extremely helpful in detecting subtle program errors resulting from type mismatches between actual and formal arguments to a function, and its use is highly recommended. However, you are not required to use argument type checking. The function declarations in the include files are enclosed in preprocessor \#if defined() blocks, and are enabled only when you define the identifier LINT_ARGS.

To provide argument-type lists for all run-time functions, several new include files have been added to the list of standard include files for the C run-time library. The names of the new include files have been chosen to maintain as much compatibility as possible with the proposed ANSI (American National Standards Institute) standard for C (Draft Proposed American National Standard—Programming Language C) and with XENIX and UNIX names.

1.2 About This Manual

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

The Microsoft C Optimizing Compiler Run-Time Library Reference has two major parts. Part 1, "Overview," gives an introduction to the C run-time library. It discusses general rules that apply to the run-time library as a whole and summarizes the elements of the run-time library.
Part 2, "Reference," gives descriptions of the run-time routines in alphabetical order for quick reference. Once you have familiarized yourself with the library rules and procedures, you will probably use the second part of the manual most often.

The remaining chapters of Part 1 are as follows:

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

Chapter 3, "Global Variables and Standard Types," describes variables and types that are defined in the run-time library and used by run-time library routines. This chapter also provides a cross-reference to the include file that defines or declares each variable or type. You may find these variables and types useful in your own routines. The variables and types are also described on the reference pages for the routines that use them in Part 2, "Reference."

Chapter 4, "Run-Time Routines by Category," breaks down the run-time library routines by category, lists the routines that fall into each category, and discusses considerations that apply to each category as a whole. The chapter is intended to complement Part 2, "Reference," making it easy to locate routines by task. Once you have located the names of the routines you want, you will need to turn to the appropriate page in Part 2 for a detailed description.

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

The appendixes, which follow Part 2, provide more detailed information about error messages and about XENIX-compatible routines. Appendix A, "Error Messages," describes the error values and messages that can appear when using library routines. Appendix B, "A Common Library for XENIX and DOS," lists routines of the DOS C library that operate compatibly with routines of the same name on XENIX (and UNIX) systems. Appendix B also describes any differences between the DOS and XENIX versions of the routines. Common global variables and include files are also discussed in this appendix.
The remainder of this chapter describes the notational conventions used throughout the manual.

### 1.3 Notational Conventions

The following notational conventions are used throughout this manual:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>C keywords, such as <code>double</code> and <code>char</code>, are set in bold type to distinguish them from ordinary identifiers and text. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown. The names of run-time library routines, include files, global variables, standard types, constants, and identifiers used by the C library are also set in this font to emphasize that these names are reserved by the run-time library. For example, the routine name <code>strcpy</code> appears in this font; so does the include file <code>stdio.h</code>.</td>
</tr>
<tr>
<td><strong>BOLD CAPITALS</strong></td>
<td>Bold capital letters are used for the names of environment variables (such as <code>TZ</code> and <code>PATH</code>) and DOS commands (such as <code>SET</code> and <code>PATH</code>). However, on DOS you are not required to use capital letters for these variables and commands.</td>
</tr>
<tr>
<td><strong>Italics</strong></td>
<td>Italics are used for the names of arguments to library routines. In an actual program, a specific name or value replaces the italicized argument name. For example, in the argument <code>string</code> is italicized to indicate that this is the general form for the <code>atof</code> routine. In an actual program, the user supplies a particular argument for the placeholder <code>string</code>. Occasionally, italics are used to emphasize particular words in the text.</td>
</tr>
</tbody>
</table>
Examples

Programming examples are displayed in a special typeface to resemble the output on your screen or the output of commonly used computer printers. Program fragments and variables quoted within regular text also appear in this format, as do error messages.

User input

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

```
Cosine value = :  
```

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

Ellipsis dots

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

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

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

```
= { expression [,expression]... }  
```

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

[Double brackets]

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

```
int open(pathname, oflag[, pmode]);  
```

the double brackets around `pmod e` indicate that this argument is optional and that, when given, `pmod e` must be separated from the previous argument by a comma.
Since the C language also uses brackets for array declarations and subscript expressions, these appear as single brackets in syntax discussions and examples containing arrays and subscript expressions. To illustrate,

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

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

Quotation marks set off terms defined in the text. For example, the term "token" appears in quotation marks when it is defined.

Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. For example,

"abc"

is a C string.

Small capital letters are used for the names of keys and key sequences, such as CONTROL-C.
Chapter 2
Using C Library Routines

2.1 Introduction 11
2.2 Identifying Functions and Macros 11
2.3 Including Files 13
2.4 Declaring Functions 14
2.5 Stack Checking on Entry 15
2.6 Argument Type Checking 16
2.7 Error Handling 18
2.8 File Names and Path Names 19
2.9 Binary and Text Modes 22
2.10 DOS Considerations 23
2.11 Floating-Point Support 25
2.12 Using Huge Arrays with Library Functions 26
2.1 Introduction

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

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

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

1. Include a given file in your program. Many routines require definitions and declarations that are provided by an include file.

2. Provide declarations for library functions that return values of any type but *int*. The compiler expects all functions to have *int* return type unless declared otherwise. You can provide these declarations by including the C library file containing the declarations or by explicitly declaring the functions within your program.

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

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

2.2 Identifying Functions and Macros

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

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

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

1. Some macros may treat arguments with side effects incorrectly when the macro is defined so that arguments are evaluated more than once. See the example that follows this list.

2. A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address, as a function identifier does. You cannot, therefore, use a macro identifier in contexts requiring a pointer. For instance, if you give a macro identifier as an argument in a function call, the value represented by the macro is passed; if you give a function identifier as an argument in a function call, the address of the function is passed.

3. Since macros are not functions, they cannot be declared, nor can pointers to macro identifiers be declared. Thus, type checking cannot be performed on macro arguments. The compiler does, however, detect cases where the wrong number of arguments is specified for the macro.

4. The library routines implemented as macros are defined through preprocessor directives in the library include files. To use a library macro, you must include the appropriate file, or the macro will be undefined.

The routines that are implemented as macros are marked with a note in Part 2, "Reference," of this manual. You can examine a particular macro definition in the corresponding include file to determine whether
arguments with side effects will cause problems.

Example

```c
#include <ctype.h>

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

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

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

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

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

### 2.3 Including Files

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

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

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

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

• Two sets of function declarations
  The first set of declarations gives return types and argument-type lists for run-time functions, while the second set declares only the return type. Declaring the return type is required for any function that returns a value with type other than _int_. (See Section 2.4, "Declaring Functions.") The presence of an argument-type list enables type checking for the arguments in a function call; see Section 2.5, "Argument Type Checking," for a discussion of this option.

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

The reference page in Part 2 for each library routine lists the include file or files needed by the routine.

## 2.4 Declaring Functions

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

Two sets of function declarations are provided in each include file. The first set declares both the return type and the argument-type list for the function. This set is included only when you enable argument type checking, as described in Section 2.5. Use of the type-checking feature is highly recommended, since type mismatches between a function's arguments and formal parameters can cause serious and possibly hard-to-detect errors.
The second set of function declarations declares only the return type. This set is included when argument type checking is not enabled.

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

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

you can also include the following declaration:

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

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

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

### 2.5 Stack Checking on Entry

Stack checking means that on entry to a routine, the stack is first checked to determine whether or not there is room for the local variables used by that routine. If there is, the stack pointer allocates this space. Otherwise, a "Stack Overflow" runtime error occurs. If stack checking is disabled, the compiler assumes there is space and simply resets the stack pointer. If in fact there is not sufficient space on the stack, you may overwrite memory locations in the data segment with no warning.

Typically, only functions with large local variable requirements (more than about 150 bytes) have stack checking enabled, since there is enough free space between the stack and data segments to handle functions with smaller requirements. Of course, if the function is called many times having stack checking enabled slows down compilation slightly.
The following library functions have stack checking enabled:

- `printf`
- `scanf`
- `spawne`
- `fprintf`
- `fscanf`
- `spawnve`
- `sprintf`
- `sscanf`
- `spawnvp`
- `vprintf`
- `spawnv`
- `system`

### 2.6 Argument Type Checking

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

For functions that you write yourself, you are responsible for setting up argument-type lists to invoke type checking. You can also use the `/Zg` command-line option to cause the compiler to generate a list of function declarations for all functions defined in a particular source file; the list can then be incorporated into your program. See Chapter 3, “Compiling,” in the *Microsoft C Optimizing Compiler User’s Guide* for details on using the `/Zg` option.

For functions in the C run-time library, you can use the procedures outlined in this section to perform type checking on arguments. Every function in the C run-time library is declared in one or more 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 `#if defined()` 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 will not be performed.

By default, `LINT_ARGS` is undefined, so no type checking is performed for library function arguments. You can define `LINT_ARGS` in one of two ways:
1. Use the /D command-line option to define **LINT_ARG**S at compile time.

2. Define **LINT_ARG**S with a `#define` directive in your source file. For the given file to be effective, the `#define` directive must occur before the `#include` directive.

The value of **LINT_ARG**S is not significant; you can define it as any value, including an empty value.

Note that the **LINT_ARG**S 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_ARG**S identifier by using an `#if` or `#if defined( )` 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`, `printf`, and `scanf`, type checking is performed on the first two arguments: the file or buffer and the format string.
- In calls to `open`, only the first two arguments are type checked: the path name and the open flag.
- In calls to `sopen`, the first three arguments are type checked: the path name, open flag, and sharing mode.
- In calls to `execl`, `execl`, `execlp`, and `execlpe`, type checking is performed on the first two arguments: the path name and the first argument pointer.
- In calls to `spawnl`, `spawnle`, `spawnlp`, and `spawnlpe`, type checking is performed on the first three arguments: the mode flag, the path name, and the first argument pointer.
2.7 Error Handling

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

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

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

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

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

The math functions set \texttt{errno} upon error in the manner described on the reference page for each math function in Part 2. Math functions handle errors by invoking a function named \texttt{matherr}. You can choose to handle math errors differently by writing your own error function and naming it \texttt{matherr}. When you provide your own \texttt{matherr} function, that function is used in place of the run-time library version. You must follow certain rules when writing your own \texttt{matherr} function, as outlined on the \texttt{matherr} reference page in Part 2, "Reference."
You can check for errors in stream operations by calling the `ferror` function. The `ferror` function detects whether the error indicator has been set for a given stream. The error indicator is cleared automatically when the stream is closed or rewound, or the `clearerr` function can be called to reset the error indicator.

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

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

### 2.8 File Names and Path Names

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

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

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

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

```c
#include <STDIO.H>
#include <stdio.h>
```
To produce portable code, you should use the name that works correctly on XENIX, since either case works on DOS.

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

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

---

**Note**

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

---

The above rule applies to most of the functions in the run-time library: wherever a path-name argument is required, you can use either a forward slash or a backslash as a delimiter. If you are concerned with portability to XENIX, you should use the forward slash.

However, the exceptions to the rule are important. The following functions accept string arguments that are not known in advance to be path names (they may be path names, but are not required to be). In these cases, the arguments are treated as C strings, and the following special rules apply:
In the **exec** and **spawn** families of functions, you pass the name of a program to be executed as a child process and then pass strings representing arguments to the child process. The path name of the program to be executed as the child process can use either forward slashes or backslashes as delimiters, since a path-name argument is expected. However, it is recommended that you use backslashes in any path-name arguments to the child process, since the program being executed as the child process may simply expect a string argument that is not necessarily a path name.

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

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

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

**Examples**

```c
/************************* Example 1 **************************/
result = system("DIR B:\TOP\DOWN");

/************************* Example 2 **************************/
spawnl(P_WAIT, "bin/show", "show", "sub", "bin\tell", NULL);
```

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

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

does not contain a path name.

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

## 2.9 Binary and Text Modes

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

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

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

- **To process a few selected files in binary mode, while retaining the default text mode for most files,** you can specify binary mode when you open the selected files. The `fopen` function opens a file in binary mode when the letter `b` is specified in the access `type` string for the file. If you use the `open` function, you can specify the `O_BINARY` flag in the `oflag` argument to cause the file to be opened in binary mode. For more information, see the reference pages for these functions in Part 2, "Reference".

- **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 `stdin`, `stdout`, and `stderr`, 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 *Microsoft C Optimizing Compiler User's Guide*.

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

You can still override the default mode (now binary) for particular files by opening them in text mode. The **fopen** function opens a file in text mode when the letter 't' is specified in the access type string for the file. If you use the **open** function, you can specify the **O_TEXT** flag in the **oflag** argument to cause the file to be opened in text mode. For more information, see the reference pages in Part 2 for these functions.

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

### 2.10 DOS Considerations

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dosexterr</strong>, <strong>locking</strong>, <strong>sopen</strong></td>
<td>These three functions are effective only on DOS versions 3.0 and later. The <strong>sopen</strong> function opens a file with file-sharing attributes; this function should be used instead of <strong>open</strong> when you want a file to have such attributes. The <strong>locking</strong> function locks all or part of a file from access by other users. The <strong>dosexterr</strong> function provides error handling for system call 59H in DOS versions 3.0 and later.</td>
</tr>
<tr>
<td><strong>dup</strong>, <strong>dup2</strong></td>
<td>In certain cases, using the <strong>dup</strong> and <strong>dup2</strong> functions on versions of DOS earlier than 3.0 may cause unexpected results. When you use <strong>dup</strong> or <strong>dup2</strong> to</td>
</tr>
</tbody>
</table>
create a duplicate file handle for `stdin`, `stdout`, `stderr`, `stdaux`, or `stdprt` under versions of DOS earlier than 3.0, calling the `close` function with either handle causes errors in later I/O operations using the other handle. Under DOS versions 3.0 and later, the `close` is handled correctly and does not cause later errors.

**exec, spawn**

When using the `exec` and `spawn` families of functions under versions of DOS earlier than 3.0, the value of the `arg0` argument (or `argv[0]` to the child process) is not available to the user; the character "C" is stored in that position instead. Under DOS version 3.0 and later, the complete command path is stored in `arg0`.

To write programs that will run on all versions of DOS, you can use the `_osmajor` and `_osminor` variables (discussed in Section 3.5 of Chapter 3, "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**

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

```c
unsigned char _osmajor;

if (_osmajor < 3)
    open("TEST.DAT", O_RDWR);
else
    sopen("TEST.DAT", O_RDWR, SH_DENYWR);
```
2.11 Floating-Point Support

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td></td>
</tr>
<tr>
<td>atan2</td>
<td></td>
</tr>
<tr>
<td>atof</td>
<td></td>
</tr>
<tr>
<td>bessel</td>
<td></td>
</tr>
<tr>
<td>cabs</td>
<td></td>
</tr>
<tr>
<td>ceil</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>fabs</td>
<td></td>
</tr>
<tr>
<td>fcvt</td>
<td></td>
</tr>
<tr>
<td>fmsbintoieee</td>
<td></td>
</tr>
<tr>
<td>_fnpreset</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td></td>
</tr>
<tr>
<td>freeetomsbin</td>
<td></td>
</tr>
<tr>
<td>frexp</td>
<td></td>
</tr>
<tr>
<td>gcvt</td>
<td></td>
</tr>
<tr>
<td>hypot</td>
<td></td>
</tr>
<tr>
<td>idexp</td>
<td></td>
</tr>
<tr>
<td>ldexp</td>
<td></td>
</tr>
<tr>
<td>log</td>
<td></td>
</tr>
<tr>
<td>log10</td>
<td></td>
</tr>
<tr>
<td>log10</td>
<td></td>
</tr>
<tr>
<td>_status87</td>
<td></td>
</tr>
<tr>
<td>pow</td>
<td></td>
</tr>
<tr>
<td>prexct</td>
<td></td>
</tr>
<tr>
<td>sinh</td>
<td></td>
</tr>
<tr>
<td>sinh</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>sqtn</td>
<td></td>
</tr>
<tr>
<td>tan</td>
<td></td>
</tr>
<tr>
<td>tanh</td>
<td></td>
</tr>
</tbody>
</table>

1 Not available with the /FPa compiler option
2 The bessel function does not correspond to a single function, but to six functions named j0, j1, jn, y0, y1, and yn.

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

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

When you use a floating-point type character in the format string for a printf or scanf call (cprintf, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, cscanf, fscanf, scanf, or sscanf), make sure that you specify floating-point values or pointers to floating-point values in the argument list to correspond to any floating-point type characters in the format string. The presence of floating-point arguments allows the compiler to detect the use of floating-point values. If a floating-point type character is used to print, for example, an integer argument, the use of floating-point values will not be detected because the compiler does not actually read the format string used in the printf and scanf functions. For instance, the following program produces an error at run time:
main() /* THIS EXAMPLE PRODUCES AN ERROR */
{
    long f = 10L;
    printf("%f", f);
}

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

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

As a result, the following error occurs:

Floating point not loaded

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

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

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

### 2.12 Using Huge Arrays with Library Functions

In programs that use the small, compact, medium, and large memory models, Microsoft C allows you to use arrays exceeding the 64K (kilobyte) limit of physical memory in these models by explicitly declaring the arrays as huge. (See Chapter 8 of the *Microsoft C Optimizing Compiler User’s Guide* “Working with Memory Models,” for a complete discussion of memory models and the near, far, and huge keywords.) However, you cannot generally pass huge data items as arguments to C library functions. In the case of small and medium models, where the default size of a data pointer is near (16 bits), the only routines that accept huge pointers are malloc and free. 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, write to, search, sort, copy, initialize, compare, or dynamically allocate and free huge arrays; any of these functions can be passed a huge pointer in a compact-, large-, or huge-model program without difficulty.
Chapter 3
Global Variables
and Standard Types

3.1 Introduction 31
3.2 _amblksiz 31
3.3 daylight, timezone, tzname 32
3.4 _doserrno, errno, sys_errlist, sys_nerr 33
3.5 _fmode 34
3.6 _osmajor, _osminor 34
3.7 environ, _psp 35
3.8 Standard Types 36
3.1 Introduction

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

3.2 _amblksiz

int _amblksiz;

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

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

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

_amblksiz = 2000;

Since the heap allocator always rounds the DOS request to the nearest power of two greater than or equal to _amblksiz, the preceding statement causes the heap allocator to reserve memory in the heap in multiples of 2048.
Note that adjusting the value of _ambksiz affects only far-heap allocation (for example, standard malloc calls in compact, large, and huge memory models and _fmalloc calls in small and medium memory models). Adjusting this value has no effect on calloc or _nmalloc in any memory model.

3.3 daylight, timezone, tzname

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

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

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

SET TZ=EST5EDT

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

SET TZ=EST5

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

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

\texttt{PST8PDT}

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

### 3.4 \_doserrno, errno, sys\_errlist, sys\_nerr

\begin{verbatim}
int _doserrno;
int errno;
char *sys_errlist[];
int sys_nerr;
\end{verbatim}

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

The \texttt{errno} values on DOS are a subset of the values for \texttt{errno} on XENIX systems. Therefore, the value assigned to \texttt{errno} in case of error does not necessarily correspond to the actual error code returned by an DOS system call. Instead, the actual DOS error codes are mapped onto the \texttt{perror} values. If you want to access the actual DOS error code, use the \_doserrno variable. When an error occurs in a system call, the \_doserrno variable is assigned the actual error code returned by the corresponding DOS system call. (See the \textit{Microsoft MS-DOS Programmer's Reference Manual} for details on DOS error returns.)
In general, you should use _doserrno only for error detection in operations involving input and output, since the errno values for input and output errors have DOS error-code equivalents. Not all of the error values available for errno have exact DOS error-code equivalents, and some may have no equivalents, causing the value of _doserrno to be undefined.

3.5 _fmode

int _fmode;

The _fmode variable controls the default file-translation mode. It is declared in stdlib.h. By default, the value of _fmode is 0, causing files to be translated in text mode (unless specifically opened or set to binary mode). When _fmode is set to O_BINARY, the default mode is binary. You can set _fmode to O_BINARY by linking with BINMODE.OBJ or by assigning it the value O_BINARY. See Section 2.8, “Binary and Text Modes,” in Chapter 2, “Using C Library Routines,” for a discussion of file-translation modes and the use of the _fmode variable.

3.6 _osmajor, _osminor

unsigned char _osmajor;
unsigned char _osminor;

The _osmajor and _osminor variables provide information about the version number of DOS currently in use. They are declared in stdlib.h. The _osmajor variable holds the “major” version number. For example, under DOS Version 3.0, _osmajor is 3.

The _osminor variable stores the “minor” version number. For example, under DOS Version 2.0, _osminor is 0 (zero), while under DOS Version 2.1, _osminor is 1.

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 fopen; if the major version number is earlier (less) than 3, fopen should be used instead of fopen.
3.7  environ, _psp

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

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

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

```
name= string
```

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

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

The `_psp` variable contains the segment address of the program segment prefix (PSP) for the process. The PSP contains execution information about the process, such as a copy of the command line that invoked the process and the return address for process terminate or interrupt. (See your Microsoft MS-DOS Programmer’s Reference Manual for details.) The `_psp` variable can be used to form a long pointer to the PSP, where `_psp` is the segment value and 0 is the offset value.
## 3.8 Standard Types

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

<table>
<thead>
<tr>
<th>Standard Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex</td>
<td>The <code>complex</code> structure, defined in <code>math.h</code>, stores the real and imaginary parts of a complex number and is used by the <code>cabs</code> function.</td>
</tr>
<tr>
<td>div_t, ldiv_t</td>
<td>The <code>div_t</code> and <code>ldiv_t</code> structures, defined in <code>stdlib.h</code>, store the values returned by the <code>div</code> and <code>ldiv</code> functions, respectively.</td>
</tr>
<tr>
<td>DOSERROR</td>
<td>The <code>DOSERROR</code> structure, defined in <code>dos.h</code>, stores values returned by the DOS system call 59H (available under DOS versions 3.0 and later).</td>
</tr>
<tr>
<td>exception</td>
<td>The <code>exception</code> structure, defined in <code>math.h</code>, stores error information for math routines and is used by the <code>matherr</code> routine.</td>
</tr>
<tr>
<td>FILE</td>
<td>The <code>FILE</code> structure, defined in <code>stdio.h</code>, is the structure used in all stream input and output operations. The fields of the <code>FILE</code> structure store information about the current state of the stream.</td>
</tr>
<tr>
<td>jmp_buf</td>
<td>The <code>jmp_buf</code> type, declared in <code>setjmp.h</code>, is an array type rather than a structure type. It defines the buffer used by the <code>setjmp</code> and <code>longjmp</code> routines to save and restore the program environment.</td>
</tr>
<tr>
<td>REGS</td>
<td>The <code>REGS</code> union, defined in <code>dos.h</code>, stores byte and word register values to be passed to and returned from calls to the DOS interface functions.</td>
</tr>
<tr>
<td>SREGS</td>
<td>The <code>SREGS</code> structure, defined in <code>dos.h</code>, stores the values of the ES, CS, SS, and DS registers. This structure is used by the DOS interface</td>
</tr>
</tbody>
</table>
functions that require segment register values (int86x, intdosx, and segread).

**stat**

The **stat** structure, defined in sys\stat.h, contains file-status information returned by the **stat** and **fstat** routines.

**timeb**

The **timeb** structure, defined in sys\timeb.h, is used by the **ftime** routine to store the current system time in a broken-down format.

**tm**

The **tm** structure, defined in time.h, is used by the **asctime**, **gmtime**, and **localtime** functions to store and retrieve time information.

**utimbuf**

The **utimbuf** structure, defined in sys\utime.h, stores file access and modification times used by the **utime** function to change file-modification dates.
Chapter 4
Run-Time Routines by Category

4.1 Introduction 41
4.2 Buffer Manipulation 41
4.3 Character Classification and Conversion 42
4.4 Data Conversion 44
4.5 Directory Control 45
4.6 File Handling 45
4.7 Input and Output 46
  4.7.1 Stream Routines 48
    4.7.1.1 Opening a Stream 50
    4.7.1.2 Predefined Stream Pointers:
        stdin, stdout, stderr, stdaux, stdprn 50
    4.7.1.3 Controlling Stream Buffering 52
    4.7.1.4 Closing Streams 52
    4.7.1.5 Reading and Writing Data 53
    4.7.1.6 Detecting Errors 53
4.7.2 Low-Level Routines 54
  4.7.2.1 Opening a File 54
  4.7.2.2 Predefined Handles 55
  4.7.2.3 Reading and Writing Data 56
  4.7.2.4 Closing Files 57
4.7.3 Console and Port I/O 57
4.8 Math 58
4.9 Memory Allocation 60
4.10 System Calls and Interrupts  64
4.10.1 DOS Interface  65
4.10.2 Processor Calls  66
4.11 Process Control  66
4.12 Searching and Sorting  70
4.13 String Manipulation  70
4.14 Time  72
4.15 Variable-Length Argument Lists  73
4.16 Miscellaneous  73
4.1 Introduction

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

4.2 Buffer Manipulation

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

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

When the source and target areas overlap, only the `memcpy` and `memmove` functions are guaranteed to properly copy the full source.

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

### 4.3 Character Classification and Conversion

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isalnum</code></td>
<td>Tests for alphanumeric character</td>
</tr>
<tr>
<td><code>isalpha</code></td>
<td>Tests for alphabetic character</td>
</tr>
<tr>
<td><code>iscntrl</code></td>
<td>Tests for control character</td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td>Tests for decimal digit</td>
</tr>
<tr>
<td><code>isgraph</code></td>
<td>Tests for printable character except space</td>
</tr>
<tr>
<td><code>ispunct</code></td>
<td>Tests for punctuation character</td>
</tr>
<tr>
<td><code>isspace</code></td>
<td>Tests for white-space character</td>
</tr>
<tr>
<td><code>isupper</code></td>
<td>Tests for uppercase character</td>
</tr>
<tr>
<td><code>isxdigit</code></td>
<td>Tests for hexadecimal digit</td>
</tr>
<tr>
<td><code>tolower</code></td>
<td>Converts character to lowercase if uppercase</td>
</tr>
<tr>
<td><code>toupper</code></td>
<td>Converts character to uppercase if lowercase</td>
</tr>
<tr>
<td><code>_tolower</code></td>
<td>Converts character to lowercase (unconditional)</td>
</tr>
<tr>
<td><code>_toupper</code></td>
<td>Converts character to uppercase (unconditional)</td>
</tr>
</tbody>
</table>
The character classification and conversion routines let you test individual characters in a variety of ways and convert between uppercase and lowercase characters. The classification routines identify a character by finding it in a table of classification codes; using these routines to classify a character is generally faster than writing a test expression such as the following:

\[
\text{if } ((c >= 0) \text{ || } c <= 0x7f))
\]

The `tolower` and `toupper` routines are implemented both as functions and as macros; the remainder of the routines in this category are implemented only as macros. All of the macros are defined in `ctype.h`, and this file must be included or the macros will be undefined.

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

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

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

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

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

43
3. Include `stdlib.h`.

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

### 4.4 Data Conversion

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atof</code></td>
<td>Converts string to <code>float</code></td>
</tr>
<tr>
<td><code>atoi</code></td>
<td>Converts string to <code>int</code></td>
</tr>
<tr>
<td><code>atol</code></td>
<td>Converts string to <code>long</code></td>
</tr>
<tr>
<td><code>ecvt</code></td>
<td>Converts <code>double</code> to string</td>
</tr>
<tr>
<td><code>fcvt</code></td>
<td>Converts <code>double</code> to string</td>
</tr>
<tr>
<td><code>gcvt</code></td>
<td>Converts <code>double</code> to string</td>
</tr>
<tr>
<td><code>itoa</code></td>
<td>Converts <code>int</code> to string</td>
</tr>
<tr>
<td><code>ltoa</code></td>
<td>Converts <code>long</code> to string</td>
</tr>
<tr>
<td><code>strtod</code></td>
<td>Converts string to <code>double</code></td>
</tr>
<tr>
<td><code>strtol</code></td>
<td>Converts string to a <code>long</code> decimal integer equal to a number with the specified radix</td>
</tr>
<tr>
<td><code>strtoul</code></td>
<td>Converts string to an <code>unsigned long</code> decimal integer equal to a number with the specified radix</td>
</tr>
<tr>
<td><code>ultoa</code></td>
<td>Converts <code>unsigned long</code> to string</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdir</td>
<td>Changes current working directory</td>
</tr>
<tr>
<td>getcwd</td>
<td>Gets current working directory</td>
</tr>
<tr>
<td>mkdir</td>
<td>Makes a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>Removes a directory</td>
</tr>
</tbody>
</table>

The directory-control routines let you access, modify, 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 directory routines are functions and are declared in the include file `direct.h`.

### 4.6 File Handling

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

The file-handling routines work on a file designated by a path name or file handle. They modify or give information about the designated file. All of these routines except fstat and stat are declared in the include file io.h. The fstat and stat functions are declared in sys\stat.h. The remove and rename functions are also declared in stdio.h.

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

The chsize, filelength, isatty, locking, setmode, and fstat routines work with files designated by a file handle. The locking routine works only under DOS versions 3.0 and later; it 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 file name. Programs can use mktemp to create unique file names that do not conflict with the names of existing files. The umask routine sets the default permission mask for any new files created in a program. The mask may override the permission setting given in the open or creat call for the new file.

4.7  Input and Output

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

1. Stream I/O
2. Low-level I/O
3. Console and port I/O
The “stream” functions treat a data file or data item as a stream of individual characters. By choosing among the many stream functions available, you can process data in different sizes and formats, from single characters to large data structures.

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

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

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

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

The “low-level” input and output routines do not perform buffering and formatting; they may be considered as invoking the operating system’s input and output capabilities directly. These routines let you access files and peripheral devices at a more basic level than the stream functions.

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

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

### 4.7.1 Stream Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr</td>
<td>Clears the error indicator for a stream</td>
</tr>
<tr>
<td>fclose</td>
<td>Closes a stream</td>
</tr>
<tr>
<td>fcloseall</td>
<td>Closes all open streams</td>
</tr>
<tr>
<td>fdopen</td>
<td>Opens a stream using a handle</td>
</tr>
<tr>
<td>feof</td>
<td>Tests for end-of-file on a stream</td>
</tr>
<tr>
<td>ferror</td>
<td>Tests for error on a stream</td>
</tr>
<tr>
<td>fflush</td>
<td>Flushes a stream</td>
</tr>
<tr>
<td>fgetc</td>
<td>Reads a character from stream (function version)</td>
</tr>
<tr>
<td>fgetchar</td>
<td>Reads a character from stdin (function version)</td>
</tr>
<tr>
<td>fgetpos</td>
<td>Read the position indicator of stream</td>
</tr>
<tr>
<td>fgets</td>
<td>Reads a string from stream</td>
</tr>
<tr>
<td>fileno</td>
<td>Gets file handle associated with stream</td>
</tr>
<tr>
<td>flushall</td>
<td>Flushes all streams</td>
</tr>
<tr>
<td>fopen</td>
<td>Opens a stream</td>
</tr>
<tr>
<td>fprintf</td>
<td>Writes formatted data to stream</td>
</tr>
<tr>
<td>fputc</td>
<td>Writes a character to stream (function version)</td>
</tr>
<tr>
<td>fputchar</td>
<td>Writes a character to stdout (function version)</td>
</tr>
<tr>
<td>fputs</td>
<td>Writes a string to stream</td>
</tr>
<tr>
<td><strong>Routine</strong></td>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>fread</td>
<td>Reads unformatted data from <em>stream</em></td>
</tr>
<tr>
<td>freopen</td>
<td>Reassigns a FILE pointer</td>
</tr>
<tr>
<td>fscanf</td>
<td>Reads formatted data from <em>stream</em></td>
</tr>
<tr>
<td>fseek</td>
<td>Repositions file pointer to given location</td>
</tr>
<tr>
<td>fsetpos</td>
<td>Set the position indicator of <em>stream</em></td>
</tr>
<tr>
<td>ftell</td>
<td>Gets current file-pointer position</td>
</tr>
<tr>
<td>fwrite</td>
<td>Writes unformatted data items to <em>stream</em></td>
</tr>
<tr>
<td>getc</td>
<td>Reads a character from <em>stream</em> (macro version)</td>
</tr>
<tr>
<td>getchar</td>
<td>Reads a character from <em>stdin</em> (macro version)</td>
</tr>
<tr>
<td>gets</td>
<td>Reads a line from <em>stdin</em></td>
</tr>
<tr>
<td>getw</td>
<td>Reads a binary int from <em>stream</em></td>
</tr>
<tr>
<td>printf</td>
<td>Writes formatted data to <em>stdout</em></td>
</tr>
<tr>
<td>putc</td>
<td>Writes a character to <em>stream</em> (macro version)</td>
</tr>
<tr>
<td>putchar</td>
<td>Writes a character to <em>stdout</em> (macro version)</td>
</tr>
<tr>
<td>puts</td>
<td>Writes a line to <em>stream</em></td>
</tr>
<tr>
<td>putw</td>
<td>Writes a binary int to <em>stream</em></td>
</tr>
<tr>
<td>rewind</td>
<td>Repositions file pointer to beginning of <em>stream</em></td>
</tr>
<tr>
<td>rmtmp</td>
<td>Removes temporary files created by tmpfile</td>
</tr>
<tr>
<td>scanf</td>
<td>Reads formatted data from <em>stdin</em></td>
</tr>
<tr>
<td>setbuf</td>
<td>Controls stream buffering</td>
</tr>
<tr>
<td>setvbuf</td>
<td>Controls stream buffering and buffer size</td>
</tr>
<tr>
<td>sprintf</td>
<td>Writes formatted data to string</td>
</tr>
<tr>
<td>sscanf</td>
<td>Reads formatted data from string</td>
</tr>
<tr>
<td>tempnam</td>
<td>Generates a temporary file name in given directory</td>
</tr>
<tr>
<td>tmpfile</td>
<td>Creates a temporary file</td>
</tr>
<tr>
<td>tmpnam</td>
<td>Generates a temporary file name</td>
</tr>
<tr>
<td>ungetc</td>
<td>Places a character in the buffer</td>
</tr>
<tr>
<td>vfprintf</td>
<td>Writes formatted data to <em>stream</em></td>
</tr>
</tbody>
</table>
vprintf          Writes formatted data to stdout  
vsprintf         Writes formatted data to a string

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

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

### 4.7.1.1 Opening a Stream

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

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

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

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

### 4.7.1.2 Predefined Stream Pointers:

stdin, stdout, stderr, stdaux, stdprn

When a program begins execution, five streams are automatically opened. These streams are the standard input, standard output, standard error, standard auxiliary, and standard print. By default, the standard input, standard output, and standard error refer to the user’s console. This means that whenever a program expects input from the “standard input,” it receives that input from the console. Similarly, a program that writes to the “standard output” prints its data to the console. Error messages generated by the library routines are sent to the “standard error,” meaning that error messages appear on the user’s console.
The assignment of the “standard auxiliary” and “standard print” streams depends on the machine configuration; these streams usually refer to an auxiliary port and a printer, respectively, but they might not be set up on a particular system. Be sure to check your machine configuration before using these streams.

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 handles:

<table>
<thead>
<tr>
<th>Handle</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>Standard input</td>
</tr>
<tr>
<td>stdout</td>
<td>Standard output</td>
</tr>
<tr>
<td>stderr</td>
<td>Standard error</td>
</tr>
<tr>
<td>stdaux</td>
<td>Standard auxiliary</td>
</tr>
<tr>
<td>stdprn</td>
<td>Standard print</td>
</tr>
</tbody>
</table>

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

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

You can redefine stdin, stdout, stderr, stdaux, or stdprn so that it refers to a disk file or to a device. The freopen routine is used for this purpose. See the freopen reference page in Part 2, "Reference", for a description of this option.

Important
At the DOS command level, stderr (standard error) cannot be redirected.

### 4.7.1.3 Controlling Stream Buffering

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

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

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

### 4.7.1.4 Closing Streams

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

The stream functions allow you to transfer data in a variety of ways. You can read and write binary data (a sequence of bytes), or specify reading and writing by characters, lines, or more complicated formats. The stream functions for reading and writing data are summarized at the beginning of this section; for a full description of each function, see Part 2, “Reference,” of this manual.

Reading and writing operations on streams always begin at the current position of the stream, known as the “file pointer” for the stream. The file pointer is changed to reflect the new position after a read or write operation takes place. For example, if you read a single character from a stream, the file pointer is increased by 1 byte so that the next operation begins with the first unread character. If a stream is opened for appending, the file pointer is automatically positioned at the end of the file before each write operation.

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

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

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

4.7.1.6 Detecting Errors

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>Closes a file</td>
</tr>
<tr>
<td>creat</td>
<td>Creates a file</td>
</tr>
<tr>
<td>dup</td>
<td>Creates a second handle for a file</td>
</tr>
<tr>
<td>dup2</td>
<td>Reassigns a file handle</td>
</tr>
<tr>
<td>eof</td>
<td>Tests for end-of-file</td>
</tr>
<tr>
<td>lseek</td>
<td>Repositions file pointer to a given location</td>
</tr>
<tr>
<td>open</td>
<td>Opens a file</td>
</tr>
<tr>
<td>read</td>
<td>Reads data from a file</td>
</tr>
<tr>
<td>sopen</td>
<td>Opens a file for file sharing</td>
</tr>
<tr>
<td>tell</td>
<td>Gets current file-pointer position</td>
</tr>
<tr>
<td>write</td>
<td>Writes data to a file</td>
</tr>
</tbody>
</table>

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

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

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

4.7.2.1 Opening a File

A file must be opened with the open, sopen, or creat function before input and output with the low-level functions can be performed on that file. The file can be opened for reading, writing, or both, and opened in either text or binary mode. The include file fcntl.h must be included when opening a file, as it contains definitions for flags used in open. In some cases the files sys\types.h and sys\stat.h must also be included; for more information see the reference page for open in Part 2 of this manual.
These functions return a file handle, to be used to refer to the file in later operations. When you call one of these functions, assign the return value to an integer variable and use that variable to refer to the opened file.

### 4.7.2.2 Predefined Handles

When a program begins execution, five file handles, corresponding to the standard input, standard output, standard error, standard auxiliary, and standard print, are already assigned. By using the following predefined handles, a program can call low-level functions to access the standard input, standard output, standard error, standard auxiliary, and standard print streams (described with the stream functions in Section 4.7.1.2).

<table>
<thead>
<tr>
<th>Stream</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>0</td>
</tr>
<tr>
<td>stdout</td>
<td>1</td>
</tr>
<tr>
<td>stderr</td>
<td>2</td>
</tr>
<tr>
<td>stdaux</td>
<td>3</td>
</tr>
<tr>
<td>stdprn</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

---

**Important**

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

---

### 4.7.2.3 Reading and Writing Data

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

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

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

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

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

4.7.3 Console and Port I/O

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>cgets</td>
<td>Reads a string from the console</td>
</tr>
<tr>
<td>cprintf</td>
<td>Writes formatted data to the console</td>
</tr>
<tr>
<td>cputs</td>
<td>Writes a string to the console</td>
</tr>
<tr>
<td>cscanf</td>
<td>Reads formatted data from the console</td>
</tr>
<tr>
<td>getch</td>
<td>Reads a character from the console</td>
</tr>
<tr>
<td>getche</td>
<td>Reads a character from the console and echoes it</td>
</tr>
<tr>
<td>inp</td>
<td>Reads specified I/O port</td>
</tr>
<tr>
<td>kbhit</td>
<td>Checks for a keystroke at the console</td>
</tr>
<tr>
<td>outp</td>
<td>Writes to specified I/O port</td>
</tr>
<tr>
<td>putch</td>
<td>Writes a character to the console</td>
</tr>
<tr>
<td>ungetch</td>
<td>“Ungets” the last character read from the console so that it becomes the next character read</td>
</tr>
</tbody>
</table>

The console and port I/O routines are implemented as functions and are declared in the include file `conio.h`. These functions perform reading and writing operations on your console or on the specified port. The `cgets`, `cscanf`, `getch`, `getche`, and `kbhit` routines take input from the console, while `cprintf`, `cputs`, `putch`, and `ungetch` write to the console. Redirecting the standard input or standard output from the command line causes the input or output of these functions to be redirected.

The console or port does not have to be opened or closed before I/O is performed, so there are no open or close routines in this category. The port I/O routines (`inp` and `outp`) read or write 1 byte at a time from the specified port. The console I/O routines allow reading and writing of strings (`cgets` and `cputs`), formatted data (`cscanf` and `cprintf`), and characters. Several options are available when reading and writing characters.
The **putch** routine writes a character to the console. The **getch** and **getche** routines read a character from the console; **getche** echoes the character back to the console, while **getch** does not. The **ungetch** routine “ungets” the last character read; the next read operation on the console begins with the “ungotten” character.

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

---

**Notes**

The console I/O routines use the corresponding low-numbered DOS system calls to read and write characters. See your "Microsoft MS-DOS Programmer’s Reference Manual" for details on the system calls.

These console routines are not compatible with stream or low-level library routines, and should not be used with them.

---

### 4.8 Math

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>Calculates arc cosine of x</td>
</tr>
<tr>
<td>asin(x)</td>
<td>Calculates arc sine of x</td>
</tr>
<tr>
<td>atan(x)</td>
<td>Calculates arc tangent of x</td>
</tr>
<tr>
<td>atan2(y,x)</td>
<td>Calculates arc tangent of y/x</td>
</tr>
<tr>
<td>bessel1</td>
<td>Calculates Bessel functions</td>
</tr>
<tr>
<td>cabs(z)</td>
<td>Finds absolute value of complex number z</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Finds integer ceiling of x</td>
</tr>
<tr>
<td>_clear87( )2</td>
<td>Gets and clears floating-point status word</td>
</tr>
<tr>
<td>_control87(new,mask)2</td>
<td>Gets old floating-point control word, and sets new control-word value</td>
</tr>
</tbody>
</table>
Run-Time Routines by Category

\[
\begin{align*}
\cos(x) & \quad \text{Calculates cosine of } x \\
cosh(x) & \quad \text{Calculates hyperbolic cosine of } x \\
dieetomsbin(&x,&y) & \quad \text{Converts IEEE double-precision number (x) to Microsoft binary format (y)} \\
dmsbintoieee(&x,&y) & \quad \text{Converts Microsoft binary double-precision number (x) to IEEE format (y)} \\
\exp(x) & \quad \text{Calculates exponential function of } x \\
fabs(x) & \quad \text{Finds absolute value of } x \\
fieetomsbin(&x,&y) & \quad \text{Converts IEEE single-precision number (x) to Microsoft binary format (y)} \\
floor(x) & \quad \text{Finds largest integer less than or equal to } x \\
fmod(x,y) & \quad \text{Finds floating-point remainder of } x/y \\
fmsbintoieee(&x,&y) & \quad \text{Converts Microsoft binary single-precision number (x) to IEEE format (y)} \\
-_fpreset() & \quad \text{Reinitializes the floating-point math package} \\
frexp(x,&n) & \quad \text{Shows } x \text{ as product of mantissa (the value returned by \texttt{frexp}) and } 2^n \\
hypot(x,y) & \quad \text{Calculates hypotenuse of right triangle with sides } x \text{ and } y \\
ldexp(x,exp) & \quad \text{Calculates } x \text{ times } 2^{\text{exp}} \\
\log(x) & \quad \text{Calculates natural logarithm of } x \\
\log10(x) & \quad \text{Calculates base 10 logarithm of } x \\
\matherr(x) & \quad \text{Handles math errors} \\
modf(x,&n) & \quad \text{Breaks down } x \text{ into integer (the value returned by \texttt{modf}) and fractional (n) parts} \\
pow(x,y) & \quad \text{Calculates } x^y \\
\sin(x) & \quad \text{Calculates sine of } x \\
\sinh(x) & \quad \text{Calculates hyperbolic sine of } x \\
sqrt(x) & \quad \text{Finds square root of } x \\
_status87()^2 & \quad \text{Gets the floating-point status word}
\end{align*}
\]
\[ \tan(x) \quad \text{Calculates tangent of } x \]
\[ \tanh(x) \quad \text{Calculates hyperbolic tangent of } x \]

1. The \texttt{bessel} routine does not correspond to a single function, but to six functions named \texttt{j0}, \texttt{j1}, \texttt{jn}, \texttt{y0}, \texttt{yl}, and \texttt{yn}.

2. Not available with the /FPa compiler option

The math routines allow you to perform common mathematical calculations. All math routines work with floating-point values, and therefore require floating-point support (see Section 2.10, “Floating-Point Support,” in Chapter 2, “Using C Library Routines”). Function declarations for the math routines are given in the include file \texttt{math.h}, with the exception of \texttt{-clear87}, \texttt{-control87}, \texttt{-fpreset}, and \texttt{-status87}, whose definitions are given in the \texttt{float.h} include file.

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

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

### 4.9 Memory Allocation

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{alloca}</td>
<td>Allocates a block of memory from the program’s stack</td>
</tr>
<tr>
<td>\texttt{calloc}</td>
<td>Allocates storage for array</td>
</tr>
<tr>
<td>\texttt{-expand}</td>
<td>Reallocates block of memory without moving its location</td>
</tr>
<tr>
<td>\texttt{-ffree}</td>
<td>FREes a block allocated by \texttt{-fmalloc}</td>
</tr>
<tr>
<td>\texttt{-fheapchk}</td>
<td>Performs a consistency check on the far heap</td>
</tr>
</tbody>
</table>
Run-Time Routines by Category

- _fheapset  Fills the far heap with a specified character
- _fheapwalk Returns information about the far heap
- _fmalloc Allocates a block of memory outside the default data segment (far heap), returns a far pointer
- free Frees a block allocated with calloc, malloc, or realloc
- _freect Returns approximate number of items of given size that could be allocated
- _fmsize Returns size of memory block pointed to by far pointer
- halloc Allocates storage for huge array
- hfree Frees a block allocated by halloc
- malloc Allocates a block
- _memavl Returns approximate number of bytes available in memory for allocation
- _memmax Returns size of largest contiguous memory location
- _msize Returns size of block allocated by calloc, malloc, or realloc
- _nfree Frees a block allocated by
- _nheapchk Performs a consistency check on the default data segment (near heap)
- _nheapset Fills the default data segment with a specified character
- _nheapwalk Returns information about the default data segment _nmalloc
_nmalloc
Allocates a block of memory in default data segment, returns a near pointer

_nmsize
Returns size of memory block pointed to by near pointer

realloc
Reallocates a block

sbrk
Resets break value

stackavail
Returns size of stack space available for allocation with alloca

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

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

The calloc and malloc routines allocate memory blocks. The malloc routine allocates a given number of bytes, while calloc allocates and initializes to 0 an array with elements of a given size. The routines _fmalloc and _nmalloc are similar to malloc, except that _fmalloc and _nmalloc allow you to allocate a block of bytes while overcoming the addressing limitations of the current memory model. The halloc routine performs essentially the same function as calloc, with the difference that halloc allocates space for huge arrays (those exceeding 64K in size).

Arrays allocated with halloc must satisfy the requirements for huge arrays discussed in Section 8.2.5 of the Microsoft C Optimizing Compiler User's Guide "Creating Huge-Model Programs."

The halloc and hfree routines differ from _nmalloc/_nfree and _fmalloc/_nfree in that the former allocate and free memory directly from DOS, instead of in the near or far heap space.

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

By contrast, when halloc is called, it allocates from DOS the amount of memory requested. It performs no heap management on the DOS memory space. When hfree is called, it simply returns the memory back to DOS.

In small data models (small- and medium-model programs), malloc maps to _nmalloc, and free maps to _nfree. In large data models (compact- and large-model programs), malloc maps to _fmalloc, and free maps to _ffree.

The _nmalloc function is fastest and should be used in small model programs where total memory allocation is less than 64K (the exact amount of memory available for near heap allocations depends on how much of the default data segment is used by the stack, program data and runtime data). The _fmalloc function is slower. It should be used where total memory allocation requirements are too large to use _nmalloc but no single data object is greater than 64k. The halloc function is the slowest of all, because it allocates from DOS for every request, but it's useful in cases where either 1) you want data objects larger than 64k, or 2) you want to make sure you can free allocated memory back to DOS for subsequent calls to spawn, etc.

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

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

The _freect and _memavl routines tell you how much memory is available for dynamic memory allocation in the default data segment; _freect returns the approximate number of items of a given size that can be allocated, while _memavl returns the total number of bytes available for allocation requests.
The \_msize function returns the size of a memory block allocated by a call to \_malloc, \_expand, \_malloc, or \_realloc. The \_fmsize and \_nmsize functions return the size of a memory block allocated by a call to \_fmalloc or \_nmalloc, respectively.

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

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

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

\section*{4.10 System Calls and Interrupts}

These routines are provided to access DOS system calls and interrupts. See your \textit{Microsoft DOS Programmer's Reference Manual} for information on system calls and interrupts.
### 4.10.1 DOS Interface

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>bdos</td>
<td>Invokes DOS system call; uses only <strong>DX</strong> and <strong>AL</strong> registers</td>
</tr>
<tr>
<td>dosexterr</td>
<td>Obtains register values from DOS system call 59H</td>
</tr>
<tr>
<td>int86</td>
<td>Invokes DOS interrupts</td>
</tr>
<tr>
<td>int86x</td>
<td>Invokes DOS interrupts</td>
</tr>
<tr>
<td>intdos</td>
<td>Invokes DOS system call; uses registers other than <strong>DX</strong> and <strong>AL</strong></td>
</tr>
<tr>
<td>intdosx</td>
<td>Invokes DOS system call; uses registers other than <strong>DX</strong> and <strong>AL</strong></td>
</tr>
</tbody>
</table>

These routines are implemented as functions and 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 provided for use with DOS versions 3.0 and later.

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

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

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

4.10.2 Processor Calls

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_OFF</td>
<td>Returns offset portion of a far pointer.</td>
</tr>
<tr>
<td>FP_SEG</td>
<td>Returns segment portion of a far pointer.</td>
</tr>
<tr>
<td>segread</td>
<td>Returns current values of segment registers.</td>
</tr>
</tbody>
</table>

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

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

4.11 Process Control

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Aborts a process</td>
</tr>
<tr>
<td>atexit</td>
<td>Executes functions at program termination</td>
</tr>
<tr>
<td>execl</td>
<td>Executes child process with argument list</td>
</tr>
<tr>
<td>execle</td>
<td>Executes child process with argument list and given environment</td>
</tr>
<tr>
<td>execlp</td>
<td>Executes child process using PATH variable and argument list</td>
</tr>
<tr>
<td>execlep</td>
<td>Executes child process using PATH variable, given environment, and argument list</td>
</tr>
<tr>
<td>execv</td>
<td>Executes child process with argument array</td>
</tr>
<tr>
<td>execve</td>
<td>Executes child process with argument array and given environment</td>
</tr>
</tbody>
</table>
execvp  Executes child process using PATH variable and argument array
execvpe Executes child process using PATH variable, given environment, and argument array
exit Terminates process
_exit Terminates process without flushing buffers
getpid Gets process ID number
raise Sends a signal to the calling process
signal Handles an interrupt signal
spawnl Executes child process with argument list
spawnle Executes child process with argument list and given environment
spawnlp Executes child process using PATH variable and argument list
spawnlpe Executes child process using PATH variable, given environment, and argument list
spawnv Executes child process with argument array
spawnve Executes child process with argument array and given environment
spawnvp Executes child process using PATH variable and argument array
spawnvpe Executes child process using PATH variable, given environment, and argument array
system Executes an DOS command

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

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

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

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

The \texttt{abort} and \texttt{\_exit} functions perform an immediate exit without flushing stream buffers. The \texttt{exit} call performs an exit after flushing stream buffers.

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

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

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

The third column describes the method for passing arguments to the child process. Passing an argument list means that the arguments to the child process are listed as separate arguments in the \texttt{exec} or \texttt{spawn} call; passing an argument array means that the arguments are stored in an array, and a pointer to the array is passed to the child process. The argument-list method is typically used when the number of arguments is constant or is known at compile time, while the argument-array method is useful when the number of arguments must be determined at run time.
The last column specifies if the child process inherits the environment settings of its parent or if a table of environment settings can be passed to set up a different environment for the child process.

### Table 4.1

**Forms of the spawn and exec Routines**

<table>
<thead>
<tr>
<th>Routines</th>
<th>Use of PATH Setting</th>
<th>Argument-Passing Convention</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>execl, spawnl</td>
<td>Do not use PATH</td>
<td>Argument list</td>
<td>Inherited from parent</td>
</tr>
<tr>
<td>execle, spawnle</td>
<td>Do not use PATH</td>
<td>Argument list</td>
<td>Pointer to environment table for child process passed as last argument</td>
</tr>
<tr>
<td>execlp, spawnlp</td>
<td>Use PATH</td>
<td>Argument list</td>
<td>Inherited from parent</td>
</tr>
<tr>
<td>execlpe, spawnlpe</td>
<td>Use PATH</td>
<td>Argument list</td>
<td>Pointer to environment table for child process passed as last argument</td>
</tr>
<tr>
<td>execv, spawnv</td>
<td>Do not use PATH</td>
<td>Argument array</td>
<td>Inherited from parent</td>
</tr>
<tr>
<td>execve, spawnve</td>
<td>Do not use PATH</td>
<td>Argument array</td>
<td>Pointer to environment table for child process passed as last argument</td>
</tr>
<tr>
<td>execvp, spawnvp</td>
<td>Use PATH</td>
<td>Argument array</td>
<td>Inherited from parent</td>
</tr>
<tr>
<td>execvpe, spawnvpe</td>
<td>Use PATH</td>
<td>Argument array</td>
<td>Pointer to environment table for child process passed as last argument</td>
</tr>
</tbody>
</table>
### 4.12 Searching and Sorting

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch</td>
<td>Performs binary search</td>
</tr>
<tr>
<td>lfind</td>
<td>Performs linear search for given value</td>
</tr>
<tr>
<td>lsearch</td>
<td>Performs linear search for given value, which is added to array if not found</td>
</tr>
<tr>
<td>qsort</td>
<td>Performs quick sort</td>
</tr>
</tbody>
</table>

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

### 4.13 String Manipulation

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcat</td>
<td>Appends a string</td>
</tr>
<tr>
<td>strchr</td>
<td>Finds first occurrence of a given character in string</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compares two strings</td>
</tr>
<tr>
<td>strcmpi</td>
<td>Compares two strings without regard to case (“i” indicates that this function is “case insensitive”)</td>
</tr>
<tr>
<td>strcpy</td>
<td>Copies one string to another</td>
</tr>
<tr>
<td>strcspn</td>
<td>Finds first occurrence of a character from given character set in string</td>
</tr>
<tr>
<td>strdup</td>
<td>Duplicates string</td>
</tr>
<tr>
<td>strerror</td>
<td>Saves system error message and optional user-error message in string</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compares two strings without regard to case (identical to <code>strcmpi</code>)</td>
</tr>
<tr>
<td>strlen</td>
<td>Finds length of string</td>
</tr>
</tbody>
</table>
Run-Time Routines by Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlwr</td>
<td>Converts string to lowercase</td>
</tr>
<tr>
<td>strncat</td>
<td>Appends $n$ characters of string</td>
</tr>
<tr>
<td>strncmp</td>
<td>Compares $n$ characters of two strings</td>
</tr>
<tr>
<td>strncpy</td>
<td>Copies $n$ characters of one string to another</td>
</tr>
<tr>
<td>strnicmp</td>
<td>Compares $n$ characters of two strings without regard to case (&quot;i&quot; indicates that this function is &quot;case insensitive&quot;)</td>
</tr>
<tr>
<td>strnset</td>
<td>Sets $n$ characters of string to given character</td>
</tr>
<tr>
<td>strpbrk</td>
<td>Finds first occurrence of character from one string in another</td>
</tr>
<tr>
<td>strrchr</td>
<td>Finds last occurrence of given character in string</td>
</tr>
<tr>
<td>strrev</td>
<td>Reverses string</td>
</tr>
<tr>
<td>strset</td>
<td>Sets all characters of string to given character</td>
</tr>
<tr>
<td>strspn</td>
<td>Finds first substring from given character set in string</td>
</tr>
<tr>
<td>strstr</td>
<td>Finds first occurrence of given string in another string</td>
</tr>
<tr>
<td>strtok</td>
<td>Finds next token in string</td>
</tr>
<tr>
<td>strupr</td>
<td>Converts string to uppercase</td>
</tr>
</tbody>
</table>

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

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

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

### 4.14 Time

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>Converts time from structure to character string</td>
</tr>
<tr>
<td>clock</td>
<td>Returns the elapsed CPU time for a process</td>
</tr>
<tr>
<td>ctime</td>
<td>Converts time from long integer to character string</td>
</tr>
<tr>
<td>difftime</td>
<td>Computes the difference between two times</td>
</tr>
<tr>
<td>ftime</td>
<td>Gets current system time as structure</td>
</tr>
<tr>
<td>gmtime</td>
<td>Converts time from integer to structure</td>
</tr>
<tr>
<td>localtime</td>
<td>Converts time from integer to structure with local correction</td>
</tr>
<tr>
<td>mktime</td>
<td>Converts time to a calendar value</td>
</tr>
<tr>
<td>time</td>
<td>Gets current system time as long integer</td>
</tr>
<tr>
<td>tzset</td>
<td>Sets external time variables from environment time variable</td>
</tr>
<tr>
<td>utime</td>
<td>Sets file-modification time</td>
</tr>
</tbody>
</table>

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

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

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

### 4.15 Variable-Length Argument Lists

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_arg</td>
<td>Retrieves argument from list</td>
</tr>
<tr>
<td>va_end</td>
<td>Resets pointer</td>
</tr>
<tr>
<td>va_start</td>
<td>Sets pointer to beginning of argument list</td>
</tr>
</tbody>
</table>

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

For more information on the differences between the two versions and for an explanation of how to use the macros, see the appropriate reference pages in Part 2 of this manual.

### 4.16 Miscellaneous

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use</th>
</tr>
</thead>
</table>
Microsoft C Compiler Run-Time Library Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Finds absolute value of integer</td>
</tr>
<tr>
<td>assert</td>
<td>Tests for logic error</td>
</tr>
<tr>
<td>div</td>
<td>Divide integers</td>
</tr>
<tr>
<td>getenv</td>
<td>Gets value of environment variable</td>
</tr>
<tr>
<td>labs</td>
<td>Finds absolute value of long integer</td>
</tr>
<tr>
<td>ldiv</td>
<td>Divide long integers</td>
</tr>
<tr>
<td>longjmp</td>
<td>Restores a saved stack environment</td>
</tr>
<tr>
<td>perror</td>
<td>Prints error message</td>
</tr>
<tr>
<td>putenv</td>
<td>Adds or modifies value of environment variable</td>
</tr>
<tr>
<td>rand</td>
<td>Gets a pseudorandom number</td>
</tr>
<tr>
<td>setjmp</td>
<td>Saves a stack environment</td>
</tr>
<tr>
<td>srand</td>
<td>Initializes pseudorandom series</td>
</tr>
<tr>
<td>swab</td>
<td>Swaps bytes of data</td>
</tr>
</tbody>
</table>

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

The abs and labs functions return the absolute value of an int and a long value, respectively. These two functions are defined in both the math.h and stdlib.h include files. (A macro named abs is also available in the include file v2tov3.h; the macro gives the absolute value for any type.)

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

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

The getenv and putenv routines provide access to the environment table. The global variable environ also points to the environment table, but it is recommended that you use the getenv and putenv routines to access and modify environment settings rather than accessing the environment table directly.
The perror routine prints the system error message, along with an optional user-supplied message, for the last system-level call that produced an error. The perror routine is declared in the include files stdlib.h and stdio.h. The error number is obtained from the errno variable. The system message is taken from the sys_errlist array. The errno variable is only guaranteed to be set upon error for those routines that explicitly mention the errno variable in the "Return Value" section of the reference pages in Part 2 of this manual.

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

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

The swab routine (also declared in stdlib.h) swaps bytes of binary data. It is typically used to prepare data for transfer to a machine that uses a different byte order.
Chapter 5
Include Files

5.1 Introduction 79
5.2 assert.h 80
5.3 conio.h 80
5.4 ctype.h 80
5.5 direct.h 81
5.6 dos.h 81
5.7 errno.h 82
5.8 fcntl.h 83
5.9 float.h 83
5.10 io.h 83
5.11 limits.h 84
5.12 malloc.h 84
5.13 math.h 84
5.14 memory.h 85
5.15 process.h 85
5.16 search.h 86
5.17 setjmp.h 86
5.18 share.h 86
5.19 signal.h 87
5.20 stdarg.h 87
5.21 stddef.h 87
5.22 stdio.h 88
5.23 stdlib.h 89
5.24 string.h 90
5.25 sys\locking.h 90
5.26 sys\stat.h 90
5.27 sys\timeb.h 91
5.28 sys\types.h 91
5.29 sys\utime.h 91
5.30 time.h 91
5.31 varargs.h 92
5.32 v2tov3.h 92
5.1 Introduction

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

A number of routines are declared in more than one include file. For example, the buffer-manipulation functions `memccpy`, `memchr`, `memcpy`, `memcmp`, `memmove`, `memset`, and `movedata` are declared in both `memory.h` and `string.h`. These multiple declarations ensure agreement with the names of XENIX and UNIX include files, as well as 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 Microsoft C.

Two sets of function declarations are provided in each include file. The first set declares both the return type and the argument-type list for the function. This set is included only when you enable argument type checking by defining `LINT_ARGS`, as described in Section 2.5, “Argument Type Checking.” The second set of declarations declares only the return type. This set is included when argument type checking is not enabled.

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

- To maintain compatibility with the names of include files on XENIX and UNIX systems, and with the ANSI standard for C
- To reflect the logical categories of run-time routines (for example, placing declarations for all memory-allocation functions in one file, `malloc.h`)
- To require the inclusion of the minimum number of files to use a given routine

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

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

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

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

5.3 conio.h

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

- cgets
- cscanf
- inp
- putch
- cprintf
- getch
- kbbhit
- ungetch
- cputs
- getche
- outp

5.4 ctype.h

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

\begin{verbatim}
isalnum  iscntrl  islower  isspace  toascii  _tolower
isalpha  isdigit  isprint  isupper  tolower  _toupper
isascii  isgraph  ispunct  isxdigit  toupper
\end{verbatim}

You must include \texttt{ctype.h} when using these macros or the macros will be undefined.

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

In addition to macro definitions, the \texttt{ctype.h} include file contains the following:

1. A set of manifest constants defined as bit masks. The bit masks correspond to specific classification tests. For example, the constants \_UPPER and \_LOWER are defined to test for an uppercase or lowercase letter, respectively.

2. A declaration of a global array, \_ctype. The \_ctype array is a table of character-classification codes based on ASCII character codes.

\section*{5.5 \texttt{direct.h}}

The \texttt{direct.h} include file contains function declarations for the four directory control functions (\texttt{chdir}, \texttt{getcwd}, \texttt{mkdir}, and \texttt{rmdir}).

\section*{5.6 \texttt{dos.h}}

The \texttt{dos.h} include file contains macro definitions, function declarations, and type definitions for the DOS interface functions.
The \texttt{FP\_SEG} and \texttt{FP\_OFF} macros are defined to get or set the segment and offset portions of a \texttt{far} pointer. You must include \texttt{dos\_h} when using these macros or they will be undefined.

The following functions are declared in \texttt{dos\_h}:

\begin{verbatim}
bdos
dosexterr
int86
int86x
intdos
intdosx
segread
\end{verbatim}

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

The \texttt{DOSERROR} structure is defined to hold error values returned by DOS system call 59H (available under DOS versions 3.0 and later).

Note that \texttt{WORDREGS}, \texttt{BYTEREGS}, \texttt{REGS}, \texttt{SREGS}, and \texttt{DOSERROR} are tags, not \texttt{typedef} names. (See the \textit{Microsoft C Optimizing Compiler Language Reference} for a discussion of type definitions, tags, and \texttt{typedef} names.)

\section*{5.7 \texttt{errno\_h}}

The \texttt{errno\_h} include file defines the values used by system-level calls to set the \texttt{errno} variable. The constants defined in \texttt{errno\_h} are used by the \texttt{perror} function to index the corresponding error message in the global variable \texttt{sys\_errlist}.

The constants defined in \texttt{errno\_h} are listed with the corresponding error messages in Appendix A, "Error Messages."
5.8 fcntl.h

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

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

5.9 float.h

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

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

In addition, `float.h` defines floating-point-exception subcodes used with `SIGFPE` to trap floating-point errors (see `signal.h` in Part 2, “Reference”).

5.10 io.h

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

```
access   dup2   mktemp   tell
chmod    eof    open     umask
chsize   filelength read   unlink
close    isatty  rename   write
creat    locking  setmode
dup      lseek   sopen
```
The exceptions are \texttt{fstat} and \texttt{stat}, which are declared in \texttt{sys/stat.h}.

\section*{5.11 \texttt{limits.h}}

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

\section*{5.12 \texttt{malloc.h}}

The include file \texttt{malloc.h} contains function declarations for the memory-allocation functions listed below:

\begin{verbatim}
alloca _fheapwalk hfree _nheapchk sbrk
calloc _fmalloc malloc _nheapset stackavail
_expand _fmsize _memmax _nheapwalk
_ffree _free _memavl _nmalloc
_fheapchk _freect _msize _nmsize
_fheapset halloc _nfree realloc
\end{verbatim}

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

\section*{5.13 \texttt{math.h}}

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

\begin{verbatim}
abs bessel1 fabs ldexp sin acos cabs floor log sinh acos cabs
asin ceil fmod log10 sinh atan cos frexp matherr sqrt atan2 cosh
atan2 cosh hypot modf tanh atof exp labs pow
\end{verbatim}
The 1beessel routine does not correspond to a single function but to six functions named \text{j0, j1, jn, y0, y1, and yn.}

The \texttt{math.h} include file also defines two structures, \texttt{exception} and \texttt{complex}. The \texttt{exception} structure is used with the \texttt{matherr} function, and the \texttt{complex} structure is used to declare the argument to the \texttt{cabs} function. Finally, the last section of the paragraph should be inverted, a la my first suggestion, and all references to \texttt{HUGE} in the man. pages, should be changed to \texttt{HUGE-VAL}.

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

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

### 5.14 memory.h

The include file \texttt{memory.h} contains function declarations for the seven buffer-manipulation routines listed below:

- \texttt{memccpy}
- \texttt{memcpy}
- \texttt{memset}
- \texttt{memchr}
- \texttt{memicmp}
- \texttt{memcmp}
- \texttt{memmove}
- \texttt{movedata}
- \texttt{memcpy}
- \texttt{memmove}

### 5.15 process.h

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

- \texttt{abort}
- \texttt{execvp}
- \texttt{spawnlp}
- \texttt{execvpe}
- \texttt{spawnlpe}
The process.h include file also defines flags used in calls to spawn functions to control execution of the child process. Whenever you use one of the eight spawn functions, you must include process.h so the flags are defined.

5.16 search.h

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

5.17 setjmp.h

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

5.18 share.h

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

The include file `signal.h` defines the values for signals. Only the SIGINT and SIGFPE (floating-point exceptions) signals are recognized on DOS. The `signal` and `raise` functions are also declared in `signal.h`.

5.20 stdarg.h

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

5.21 stddef.h

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>The null pointer (also defined in <code>stdio.h</code>)</td>
</tr>
<tr>
<td>errno</td>
<td>A global variable containing an error message number (also defined in <code>errno.h</code>)</td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>Synonym for the type (<code>int</code>) of the difference of two pointers</td>
</tr>
<tr>
<td>size_t</td>
<td>Synonym for the type (<code>int</code>) of the value returned by <code>sizeof</code></td>
</tr>
</tbody>
</table>
5.22 stdio.h

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr</td>
<td>flushall</td>
</tr>
<tr>
<td>fclose</td>
<td>fopen</td>
</tr>
<tr>
<td>fcloseall</td>
<td>fprintf</td>
</tr>
<tr>
<td>feof</td>
<td>fputc</td>
</tr>
<tr>
<td>ferror</td>
<td>fputchar</td>
</tr>
<tr>
<td>fflush</td>
<td>fread</td>
</tr>
<tr>
<td>fgetc</td>
<td>freopen</td>
</tr>
<tr>
<td>fgetpos</td>
<td>fscanf</td>
</tr>
<tr>
<td>fgets</td>
<td>fseek</td>
</tr>
</tbody>
</table>

1 Implemented as a macro.

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFSIZ</td>
<td>Buffers used in stream I/O are required to have a constant size, which is defined by the BUFSIZ constant. This value is used to establish the size of system-allocated buffers, and must also be used when calling setbuf to allocate your own buffers.</td>
</tr>
<tr>
<td>_NFILE</td>
<td>The _NFILE constant defines the number of open files allowed at one time. The files stdin, stdout, stderr, stdaux, and stdprn are always open, so you should include them when calculating the number of files your program opens.</td>
</tr>
<tr>
<td>EOF</td>
<td>The EOF value is defined to be the value returned by an I/O routine when the end of the file (or in some cases, an error) is encountered.</td>
</tr>
<tr>
<td>NULL</td>
<td>The NULL value is the null-pointer value. It is defined as 0 in small- and medium-model programs and as 0L in large-model programs.</td>
</tr>
</tbody>
</table>

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

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

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

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

### 5.23 stdlib.h

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

<table>
<thead>
<tr>
<th>atexit</th>
<th>div</th>
<th>itoa</th>
<th>rand</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>ecvt</td>
<td>labs</td>
<td>realloc</td>
<td>tolower</td>
</tr>
<tr>
<td>abs</td>
<td>exit</td>
<td>ldiv</td>
<td>srand</td>
<td>toupper</td>
</tr>
<tr>
<td>atof</td>
<td>fcvt</td>
<td>ltoa</td>
<td>strtod</td>
<td>ultoa</td>
</tr>
<tr>
<td>atoi</td>
<td>free</td>
<td>malloc</td>
<td>strtol</td>
<td></td>
</tr>
<tr>
<td>atol</td>
<td>gcvt</td>
<td>perror</td>
<td>strtoul</td>
<td></td>
</tr>
<tr>
<td>calloc</td>
<td>getenv</td>
<td>putenv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>doserrno</th>
<th>errno</th>
<th>osmajor</th>
<th>osmode</th>
<th>sys_errlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>environ</td>
<td>fmode</td>
<td>osminor</td>
<td>psp</td>
<td>sys_nerr</td>
</tr>
</tbody>
</table>
### 5.24 string.h

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcpy</td>
<td>movedata</td>
<td>strdup</td>
<td>strncpy</td>
<td>strncpy</td>
<td>strspn</td>
</tr>
<tr>
<td>memcmp</td>
<td>strcat</td>
<td>strerror</td>
<td>strnicmp</td>
<td>strnset</td>
<td>strstr</td>
</tr>
<tr>
<td>memchr</td>
<td>strchr</td>
<td>strcmp</td>
<td>strnicmp</td>
<td>strrchr</td>
<td>strtok</td>
</tr>
<tr>
<td>memcmp</td>
<td>strcmpi</td>
<td>strlen</td>
<td>strpbrk</td>
<td>strrev</td>
<td>strset</td>
</tr>
<tr>
<td>memmove</td>
<td>strcpy</td>
<td>strlwr</td>
<td>strset</td>
<td>strrev</td>
<td>strset</td>
</tr>
<tr>
<td>memset</td>
<td>strcspn</td>
<td>strncmp</td>
<td>strset</td>
<td>strrev</td>
<td>strset</td>
</tr>
</tbody>
</table>

### 5.25 sys\locking.h

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

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

### 5.26 sys\stat.h

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

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

5.28  sys\types.h

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

5.29  sys\utime.h

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

5.30  time.h

The time.h include file declares the time functions asctime, clock, ctime, difftime, gmtime, localtime, mktime, time, and tzset. (The ft ime and ut ime functions are declared in sys\timeb.h and sys\utime.h, respectively.)

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

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

5.32  v2tov3.h

The include file v2tov3.h is provided for users who are converting from versions 2.03 and earlier of the Microsoft C Compiler. Some of the routines provided in the Version 2.03 run-time library are supported in a slightly different form under Version 3.0 of the compiler. Including v2tov3.h allows those routines to be used in their original form without altering the source code.

The v2tov3.h file, as well as other differences between Version 3.0 of the Microsoft C Compiler and other versions, is discussed in detail in Appendix F, "Converting from Previous Versions of the Compiler," in the Microsoft C Optimizing Compiler User's Guide.

The v2tov3.h file contains three macro definitions that can be useful. The abs macro produces the absolute value of its argument. The min and max macros calculate the minimum and maximum, respectively, of two numbers. See the v2tov3.h include file for details.
Part 2
Reference
abort

- Summary

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

void abort();
```

- Description

The **abort** function prints the message

Abnormal program termination

to **stderr**, then terminates the calling process, returning control to the
process that initiated the calling process (usually the operating system).
The **abort** function does not flush stream buffers or do **atexit** processing.

- Return Value

An exit status of 3 is returned to the parent process or operating system.

- See Also

**excl, execle, execlp, execlepe, execv, execve, execvp, execvpe, exit, exit, signal, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe**


Example

```c
#include <stdio.h>

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

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

Sample command line:

update employ.dat

Output:

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

Abnormal program termination

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

- **Summary**

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

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

- **Description**

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

- **Return Value**

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

- **See Also**

`cabs, fabs, labs`

- **Example**

```c
#include <stdlib.h>

int x = -4, y;

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

Output:

```
-4 4
```

This program computes and displays the absolute value of -4.
# Summary

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

int access(pathname, mode);
char *pathname;  // File or directory path name
int mode;       // Permission setting
```

## Description

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>Check for read and write permission</td>
</tr>
<tr>
<td>04</td>
<td>Check for read permission</td>
</tr>
<tr>
<td>02</td>
<td>Check for write permission</td>
</tr>
<tr>
<td>00</td>
<td>Check for existence only</td>
</tr>
</tbody>
</table>

Under DOS, all existing files have read access; thus the modes 00 and 04 produce the same result. Similarly, the modes 06 and 02 are equivalent, since write access implies read access on DOS.

With directories, `access` determines only whether or not the specified directory exists; under DOS, all directories have read and write access.

## Return Value

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Access denied: the file's permission setting does not allow the specified access.</td>
</tr>
</tbody>
</table>
access

ENOENT File or path name not found.

- See Also

chmod, fstat, open, stat

- Example

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

int fh;

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

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

- Summary

```
#include <math.h>

double acos(x);

double x;
```

- Description

The **acos** function returns the arccosine of \( x \) in the range 0 to \( \pi \). The value of \( x \) must be between -1 and 1.

- Return Value

The **acos** function returns the arccosine result. If \( x \) is less than -1 or greater than 1, **acos** sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

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

- See Also

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

Example

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

extern int errno;

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

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

# include `<malloc.h>`

Required only for function declarations

char *alloca(size);

Bytes to be allocated from stack

- **Description**

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

- **Return Value**

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

- **See Also**

`calloc`, `malloc`, `realloc`

---

**Warning**

The pointer value returned by `alloca` should never be passed as an argument to `free`. Also, because `alloca` manipulates the stack, it should be used only in simple assignment statements and never in an expression that is an argument to a function.
alloca

- Example

```
#include <malloc.h>

int *intarray;

/* Allocate space on the stack for 10 integers: */
intarray = (int *)alloca(10*sizeof(int));
```

This example calls `alloca` to allocate enough bytes for 10 integers. Using the `sizeof` expression ensures portability.
# Summary

```c
#include <time.h>

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

## Description

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

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

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

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

## Return Value

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

## See Also

`ctime`, `ftime`, `gmtime`, `localtime`, `time`, `tzset`

### Note

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

Example

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

struct tm *newtime;
long clock;

main()
{
    time(&clock); /* get time in seconds */
    /* Convert time to struct tm: */

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

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

```c
#include <math.h>

double asin(x);
double x;
```

- Description

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

- Return Value

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

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

- See Also

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

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

extern int errno;

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

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

- Summary

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

void assert(expression);
```

- Description

The `assert` routine prints a diagnostic message and terminates the calling process if `expression` is false (0). The diagnostic message has the form

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

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

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

- Return Value

There is no return value.

---

**Note**

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

• Example

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

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

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

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

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

```c
#include <math.h>

double atan(x);    // Calculate arctangent of x
double x;

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

## Description

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

## Return Value

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

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

## See Also

`acos`, `asin`, `cos`, `matherr`, `sin`, `tan`

## Example

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

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

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

Output:

0.7853982
-0.7853982
Summary

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

int atexit(func);
void *func();            Function to be called

Description

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

Return Value

The atexit function returns a pointer to the function if successful, and returns NULL if there is no space left to store the function pointer.

See Also

exit
atexit

**Example**

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

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

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

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

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

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

Output:

This is executed first.
This is executed next.

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

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

double atof(string);
const char *string;

Use either math.h or stdlib.h

Convert string to double
String to be converted

#include <stdlib.h>

Required only for function declarations

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

Convert string to int
Convert string to long
String to be converted

Description

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

The atof function expects string to have the following form:

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

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

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

[whitespace][sign]digits

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

■ Return Value

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

■ See Also

ewcvt, fcvt, gcvt

■ Example

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

extern long atol();
main()
{
  char *s; double x; int i; long l;
  s = " -2309.12E-15"; /* first test of "atof" */
  x = atof(s);
  printf("%e\t",x);

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

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

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

Output:

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

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

- Summary

```c
#include <dos.h>

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

- Description

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

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

- Return Value

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

- See Also

`intdos`, `intdosx`

---

**Warning**

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

#include <dos.h>
{
    char *buffer = "Enter file name:";
    /* Call 9 prints a string terminated by
     * AL is not needed, so 0 is used */

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

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

- **Summary**

```c
#include <math.h>

double j0(x);

double j1(x);

double jn(n, x);

double y0(x);

double y1(x);

double yn(n, x);

double x;              // Floating-point value
int n;                 // Integer order
```

- **Description**

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

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

- **Return Value**

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

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

Error handling can be modified by using the `matherr` routine.
See Also

matherr

Example

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

main()
{
    double x, y, z;
    x = 2;
    y = j0(x);
    z = yn(3,x);
    printf(\"y = %f and z = %f\", y, z);
}

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

- Summary

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

void *bsearch(key, base, num, width, compare);
const void *key;       // Search key
const void *base;      // Pointer to base of search data
size_t num, width;    // Number and width of elements
int *compare(const void *elem1, const void *elem2);
    // user-defined compare function
```

- Description

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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>element1</code> less than <code>element2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>element1</code> identical to <code>element2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>element1</code> greater than <code>element2</code></td>
</tr>
</tbody>
</table>

- Return Value

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

lfind, lsearch, qsort

Example

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

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

main (argc, argv)
int argc;
char **argv;
{
    char **result;
    char *key = "PATH";
    int i;
    /* Sort using Quicksort algorithm: */
    qsort((char *)argv, argc, sizeof(char *), qcompare);
    for (i=0;i<argc;++i)
        /* Output sorted list: */
        printf("%s\n", argv[i]);

    /* Find item that begins with "PATH" */
    /* using a binary search algorithm: */
    result = (char **)bsearch((char *)&key, (char *)argv, argc,
                sizeof(char *), bcompare);

    if (result)
        printf("%s found\n", *result);
    else
        printf("PATH not found!\n");
}

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

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

This program reads the command-line parameters and sorts them using qsort. Next, the program uses bsearch to search for the parameter starting with PATH.
# Summary

```c
#include <math.h>

double cabs(z);
struct complex z;  // Contains real and imaginary parts
```

## Description

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` as follows:

```c
struct complex {
    double x, y;
};
```

A call to `cabs` is equivalent to the following:

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

## Return Value

The `cabs` function returns the absolute value as described above. On overflow, the function calls the `matherr` routine, returns the value `HUGE`, and sets `errno` to `ERANGE`.

## See Also

`abs`, `fabs`, `labs`

## Example

```c
#include <math.h>
#include <stdio.h>
main() {
    struct complex value;
    double d;

    value.x = 3.0;
    value.y = 4.0;
```
\texttt{cabs}

\begin{verbatim}
d = cabs(value);
    printf("The absolute value of the complex number 'value' is \%f", d);
\}
\end{verbatim}

This program uses \texttt{cabs} to assign the absolute value of the complex number \texttt{value} to \texttt{d}. 
### Summary

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

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

- **Description**

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

- **Return Value**

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

- **See Also**

`free, halloc, hfree, malloc, realloc`

- **Example**

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

long *lalloc;

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

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

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

```c
#include <math.h>

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

## Description

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

## Return Value

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

## See Also

`floor`, `fmod`

## Example

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

main()
{
    double y; /* y is equal to 2.0 */
    y = ceil(1.05);
    printf("The ceil( 1.05) is \%f\n",y);

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

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

- **Summary**

```c
#include <conio.h>  
char *cgets(str);
char *str;  
```

Required only for function declarations

Storage location for data

- **Description**

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

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

- **Return Value**

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

- **See Also**

getch, getche
# Example

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

char buffer[82];
char *result;

main()
{
    buffer[0] = 80; /* Maximum number of characters */
    printf("Input line of text, followed by carriage return:\n ");
    result = cgets(buffer); /* Input a line of text */
    printf("\nLine length = %d\nText = %s\n", buffer[1], result);
}

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

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

- **Summary**

```c
#include <direc.h>  // Required only for function declarations

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

- **Description**

The `chdir` function changes the current working directory to the directory specified by `pathname`. The `pathname` must refer to an existing directory.

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

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

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

- **Return Value**

The `chdir` function returns a value of 0 if the working directory is successfully changed. A return value of -1 indicates an error; in this case `errno` is set to `ENOENT`, indicating that the specified path name could not be found. No error occurs if `pathname` specifies the current working directory.

- **See Also**

`mkdir, rmdir, system`

- **Example**

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

main(argc, argv)
```
int argc;
char *argv[];
{
    int rtnval;
    if (rtnval = chdir(argv[1]))
        printf("Problem changing to directory \%s", argv[1]);
    else
        printf("Change to directory \%s was successful", argv[1]);
}

This program uses chdir to emulate the DOS cd command.
### chmod

**Summary**

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
```

Required only for function declarations

```c
int chmod(pathname, pmode);
```

Path name of existing file

```c
char *pathname;
```

Permission setting for file

```c
int pmode;
```

**Description**

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S_IWRITE</code></td>
<td>Writing permitted</td>
</tr>
<tr>
<td><code>S_IREAD</code></td>
<td>Reading permitted</td>
</tr>
<tr>
<td>`S_IREAD</td>
<td>S_IWRITE`</td>
</tr>
</tbody>
</table>

If write permission is not given, the file is made 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.

**Return Value**

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

access, creat, fstat, open, stat

Example

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

int result;
int savestderr;

main()
{
    /* make file read only: */
    result = chmod("data", S_IREAD);
    if (result == -1)
        perror("Can't change file mode");
    else
        printf("Mode changed successfully");
}
```

This program uses `chmod` to change the mode of the file `data` to read only. It then displays a message indicating whether or not the mode was changed successfully.
chsize

- Summary

```
#include <io.h>

int chsize(handle, size);
```

Required only for function declarations

- Description

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

- Return Value

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Under DOS version 3.0 and later, EACCES indicates a locking violation (the specified file is locked against access).</td>
</tr>
<tr>
<td>EBADF</td>
<td>Specified file is read only, or an invalid file handle.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>No space left on device.</td>
</tr>
</tbody>
</table>

- See Also

close, creat, open
Example

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

#define MAXSIZE 32768L

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

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

    for (i = 0; i < 50; i++)
        result = write(fh, buffer, nbytes);

    result = -1;

    if (lseek(fh,OL,2) > MAXSIZE) /* Make sure the file is */
        /* no longer than 32k */
        result = chsize(fh,MAXSIZE); /* before changing the size. */

    if (result == 0)
        printf("Size successfully changed");
    else
        printf("Problem in changing the size");
}
```

This program opens the file named DATA and writes data to it. Then it uses chsize to extend the size of DATA. Finally, it displays a message indicating whether or not the file size was successfully extended.
_clear87

- Summary

#include <float.h>

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

- Description

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

- Return Value

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

---

**Note**

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

---

- See Also

_ control87, _status87
Example

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

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

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

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

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

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

■ Summary

#include <stdio.h>

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

■ Description

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

■ See Also

eof, feof, ferror, perror

■ Example

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

FILE *stream;
int c;

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

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

#include <time.h>

clock_t clock()

• Description

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

• Return Value

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

• See Also

time, difftime

• Example

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

main()
{
    int t;
    clock_t clock();
    if (t = clock() != (clock_t - 1))
        printf("Processor time equals %d seconds\n", t/CLK_TCK);
    else
        printf("Processor time not available\n");
}

This program requests its elapsed processor time and prints it on the screen if it's available.
close

- **Summary**

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

int close(handle);
int handle;     // Handle referring to open file
```

- **Description**

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

- **Return Value**

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

- **See Also**

`chsize`, `creat`, `dup`, `dup2`, `open`, `unlink`

- **Example**

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

main()
{
  int result, fh;

  fh = open("data",O_RDONLY); /* Open the file */
  result = close(fh);         /* Now close it */
  /* Report on results: */
  if (result)
    printf("Invalid file handle argument\n");
  else
    printf("File successfully closed\n");
}
```
This program uses **open** to open a file named DATA, then uses **close** to close it.
_control87

Summary

#include <float.h>

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

Description

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

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

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

where fpcntrl is the floating-point control word.

Return Value

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

See Also

_clear87, _status87
Example

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

double a = .1;

main()
{

    /* get control word: */
    printf("control = %.4x\n", control87(0, 0));
    printf("a*a = .01 = %.15e\n", a*a);

    /* set precision to 24 bits: */
    control87(PC_24, MCW_PC);
    printf("a*a = .01 (rounded to 24 bits) = %.15e\n", a*a);

    /* restore to initial default: */
    control87(CW_DEFAULT, 0xffff);
    printf("a*a = .01 = %.15e\n", a*a);
}
```

This program uses `control87` to output the control word, set the precision to 24 bits, and reset the status to the default.
cos - cosh

- Summary

#include <math.h>

double cos(x); Calculate cosine of x
double cosh(x); Calculate hyperbolic cosine of x
double x; Radians

- Description

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

- Return Value

The cos function returns the cosine of x. If x is large, a partial loss of significance in the result may occur. In such cases, cos generates a PLOSS error, but no message is printed. 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, errno is set to ERANGE.

The cosh function returns the hyperbolic cosine of x. If the result is too large, cosh returns the value HUGE and sets errno to ERANGE. Error handling can be modified by using the matherr routine.

- See Also

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

- Example

#define PI 3.14159265359

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

main()
{
   double x,y;

This program displays the cosine and hyperbolic cosine of $\pi$. 

```c
x = cos(PI);      /* x = -1 */
y = cosh(PI);      /* y = 11.591953 */

printf("The cos(PI) = %f\n", x);
printf("The cosh(PI) = %f\n", y);
```

cprintf

- Summary

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

Required only for function declarations

```c
int cprintf(format-string[, argument...]);
```

Format control string

- Description

The `cprintf` function formats and prints a series of characters and values directly to the console, using the `putch` function to output characters. Each argument (if any) is converted and output according to the corresponding format specification in 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.

- Return Value

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

- See Also

`fprintf`, `printf`, `sprintf`, `vprintf`

---

Note

Unlike the `fprintf`, `printf`, and `sprintf` functions, `cprintf` does not translate line-feed (LF) characters into carriage-return–line-feed combinations (CR-LF) on output.
Example

#include <conio.h>

int i = -16, j = 29;
unsigned int k = 511;

main()
{
    cprintf("i=%d, j=%#x, k=%u\n", i, j, k);
    /\* Output: i=-16, j=0x1d, k=511 */
}

This program prints the values of the variables i, j, and k to the console. (The cprintf function is similar to the printf function except that it sends output to the console.)
cputs

- Summary

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

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

- Description

The `cputs` function writes the null-terminated string pointed to by `str` directly to the console. Note that a carriage-return-line-feed combination (CR-LF) is not automatically appended to the string after writing.

- Return Value

There is no return value.

- See Also

`putch`

- Example

```c
#include <conio.h>

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

main()
{
    cputs(buffer);
}
```

This program displays the prompt pointed to by `buffer` on the console.
Summary

#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>

int creat(pathname, pmode);
char *pathname;
int pmode;

Description

The creat function either creates a new file or opens and truncates an existing file. If the file specified by pathname does not exist, a new file is created with the given permission setting and opened for writing. If the file already exists and its permission setting allows writing, creat truncates the file to length 0, destroying the previous contents, and opens it for writing.

The permission setting, pmode, applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. The integer expression pmode contains one or both of the manifest constants S_IWRITE and S_IREAD, defined in sys/stat.h. When both constants are given, they are joined with the bitwise OR operator (|). The meaning of the pmode argument is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IWRITE</td>
<td>Writing permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>Reading permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IWRITE</td>
</tr>
</tbody>
</table>

If write permission is not given, the file is read only. Under DOS it is not possible to give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE are equivalent. Under DOS Version 3.0 and later, files opened using creat are always opened in compatibility mode (see sopen).

The creat function applies the current file-permission mask to pmode before setting the permissions (see umask).
creat

- Return Value

The creat function returns a handle for the created file if the call is successful. A return value of -1 indicates an error, and errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Path name specifies an existing read-only file or specifies a directory instead of a file.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>No more file handles available (too many open files).</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Path name not found.</td>
</tr>
</tbody>
</table>

- See Also

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

---

Note

The creat routine is provided primarily for compatibility with previous libraries. A call to open with the O_CREAT and O_TRUNC values specified in the oflag argument is equivalent and is preferable for new code.

---

- Example

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
    int fh;
    fh = creat("data", S_IREAD|S_IWRITE);
    if (fh == -1)
```
creatable("Couldn't create data file");
else
    printf("Created data file.\n");
}

This program uses creat to create the new file (or truncate the existing file) named DATA and open the file for writing.
cscanf

- Summary

```c
#include <conio.h>

int cscanf(format-string[, argument...]);

char *format-string;  // Format control string
```

- Description

The `cscanf` function reads data directly from the console into the locations given by the arguments (if any), using the `getche` function to read characters. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in 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`.

- Return Value

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

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

- See Also

`fscanf`, `scanf`, `sscanf`
Example

```c
#include <conio.h>

int result;
char buffer[20];

main()
{
    printf("Please enter file name: ");

    /* Read in user response; return # of matches: */
    result = cscanf("%19s",buffer);

    printf("\nNumber of correctly matched input \nitems = %d\n", result);
}
```

This program prompts for a file name and uses `cscanf` to read in the corresponding file. Then `cscanf` returns the number of items matched, and the program displays that number.
# Include <time.h>    Required only for function declarations

char *ctime(time);    Pointer to stored time

## Description

The **ctime** function converts a time stored as a long value to a character string. The *time* value is usually obtained from a call to **time**, which returns the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

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

Mon Jan 02 02:03:55 1980

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

Under DOS, dates prior to 1980 are not understood. If *time* represents a date before January 1, 1980, **ctime** returns the character string representation of 00:00:00 January 1, 1980.

## Return Value

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

## See Also

asctime, ftime, gmtime, localtime, time
Note

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

Example

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

long ltime;

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

This program gets the current time in long-integer form, then uses `ctime` to display the time in string form.
Summary

#include <math.h>

int dieeetomsbin(src8, dst8); // IEEE double to MS binary double
int dmsbintoeieee(src8, dst8); // MS binary double to IEEE double

double *src8, *dst8;

Description

The dieeetomsbin routine converts a double-precision number in IEEE format to Microsoft binary format. The dmsbintoeieee routine converts a double-precision number in Microsoft binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random-access data files created with Microsoft BASIC (which stores floating-point numbers in the Microsoft binary format), and vice versa.

The argument src8 is a pointer to the double value to be converted. The result is stored at the location given by dst8.

Return Value

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

See Also

fieeetomsbin, fmsbintoeieee

Note

These routines do not handle IEEE NANs and infinities. IEEE denormals are treated as 0 in the conversions.
difftime

- Summary

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

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

- Description

The **difftime** function computes the difference $time2 - time1$.

- Return Value

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

- See Also

**time**
Example

```c
#include <time.h>

int mark[10000];

int main()
{
    time_t start, finish;
    register int i, loop, n, num, step;
    printf("This program will take about 3 minutes \n on an AT and 8 on a PC\n");
    printf("Working...\n");

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

    /* Prints average of 1000 loops through "sieve": */
    printf("\nProgram takes %f seconds to find %d primes.\n",
            difftime(finish, start)/1000, num);
}
```

Output:

Program takes 0.482000 seconds to find 1228 primes.

This program calculates the amount of time needed to find the prime numbers between 3 and 10,000. To display the prime numbers, delete the outermost loop and the comment delimiters around printf("%d", n);.
div

■ Summary

#include <stdlib.h>

div_t div(numer, denom);
int numer; Numerator
int denom; Denominator

■ Description

The div function divides numer by denom, computing the quotient and the remainder. Where the division cannot be performed exactly, the quotient is returned with the proper sign and it's value is the largest integer value less than the exact mathematical value. The behavior of div is undefined if the value cannot be represented.

■ Return Value

The div function returns a structure of type div_t, comprising both the quotient and the remainder. The structure contains the following members:

int quot; Quotient
int rem; Remainder

■ See Also

div

■ Example

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

main(argc, argv)
  int argc;
  char **argv;
{
  int x,y;
  div_t div_result;
  x = atoi(argv[1]);
y = atoi(argv[2]);
printf("x is %d, y is %d, x, y):
div_result = div(x, y);
printf("The quotient is %d, and the remainder is %d,
   div_result.quot, div_result.rem):
}

Assuming the executable is named "tdiv", a use of it might look like:

    tdiv 5 2

and would produce this output:

    x is 5, y is 2
    The quotient is 2, and the remainder is 1

This program accepts two arguments on the command line following the
program name, then calls div to divide the first argument by the second.
Finally, it prints the structure members quot and rem.
dosexterr

- **Summary**

```c
#include <dos.h>

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

- **Description**

The *dosexterr* function obtains the register values returned by the DOS system call 59H and stores the values in the structure pointed to by `buffer`. This function is useful when making system calls under DOS Version 3.0 or later, which offers extended error handling. See your *Microsoft MS-DOS Programmer’s Reference Manual* for details on DOS system calls.

The structure type *DOSErrOR* is defined in *dos.h* as follows:

```c
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.

- **Return Value**

The *dosexterr* function returns the value in the **AX** register (identical to the value in the `exterror` structure field).

- **See Also**

`perror`
Note

The dosexterr function should be used only under DOS Version 3.0 or later.

Example

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

struct DOSERROR doserror;
int fd;

main()
{
    if ((fd = open("test.dat", O_RDONLY)) == -1)
    {
        dosexterr( &doserror );
        printf("error=%d, class=%d, action=%d, locus=%d\n",
                doserror.exterror, doserror.class, doserror.action,
                doserror.locus);
    }
    else
        printf("Open succeeded so no extended information printed");
}
```

This program tries to open the file TEST.DAT. If the attempted open operation fails, the program uses dosexterr to display extended error information.
**dup – dup2**

**Summary**

```c
#include <io.h>  

int dup(handle);
int handle;

int dup2(handle1, handle2);
int handle1;
int handle2;
```

Required only for function declarations

Create second handle for open file

Handle referring to open file

Force `handle2` to refer to `handle1` file

Handle referring to open file

Any handle value

**Description**

The `dup` and `dup2` functions cause a second file handle to be associated with a currently open file. Operations on the file can be carried out using either file handle, since all handles associated with a given file use the same file pointer. The type of access allowed for the file is unaffected by the creation of a new handle.

The `dup` function returns the next available file handle for the given file. The `dup2` function forces 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.

**Return Value**

The `dup` function returns a new file handle. The `dup2` function returns 0 to indicate success. Both functions return -1 if an error occurs, and set `errno` to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>EMFILE</td>
<td>No more file handles available (too many open files)</td>
</tr>
</tbody>
</table>
dup – dup2

■ See Also

close, creat, open

■ Example

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

int old;
FILE *new;

main()
{
    old = dup(1);   /* "old" now refers to "stdout" */
    /* Note: file handle 1 == "stdout" */
    if (old == -1)
    {
        perror("dup(1) failure");
        exit(1);
    }

    write(old, "This goes to stdout first\n", 27);

    if ((new = fopen("data", "w")) == NULL)
    {
        puts("Can't open file "data"\n");
        exit(1);
    }
    /* stdout now refers to file "data" */

    if (-1 == dup2(fileno(new), 1))
    {
        perror("Can't dup2 stdout");
        exit(1);
    }

    puts("This goes to file "data"\n");

    fflush(stdout);   /* Flush stdout stream buffer so
                     it goes to correct file */

    fclose(new);

    dup2(old, 1);    /* Restore original stdout */

    puts("This goes to stdout");
}
```
dup – dup2

This program uses the variable old to save the original stdout. It then opens a new file named NEW and forces stdout to refer to it. Finally, it restores stdout to its original state.
# Summary

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

Required only for function declarations

```c
char ecvt(value, ndigits, decptr, signptr);
```

Number to be converted

```c
double value;
int ndigits;
pnt *decptr;
```

Number of digits stored

Pointer to stored decimal point point

## Description

The `ecvt` function converts a floating-point number to a character string. The `value` is the floating-point number to be converted. `Ecvt` stores `ndigits` digits of `value` as a string and appends 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. The position of the decimal point and the sign of `value` can be obtained after the call from `decptr` and `signptr`. The argument `decptr` points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The argument `signptr` points to an integer indicating the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

## Return Value

The `ecvt` function returns a pointer to the string of digits. There is no error return.

## See Also

`atof`, `atoi`, `atol`, `fcvt`, `gcvt`
**Note**

The `ecvt` and `fcvt` functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

### Example

```c
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

main()
{
    /* buffer will contain "3141592654"
     ** decimal = 1, sign = 0
     *
     */
    buffer = ecvt(3.1415926535, precision, &decimal, &sign);
    printf("buffer= "%s"", decimal = %d, sign = %d
" buffer, decimal, sign);
}
```

This program uses `ecvt` to convert the constant 3.141592654 from a floating-point number to a character string. It then displays the resulting string.
# Summary

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

int eof(handle);
int handle;  Handle referring to open file
```

## Description

The `eof` function determines whether the end-of-file has been reached for the file associated with `handle`.

## Return Value

The `eof` function returns the value 1 if the current position is end-of-file, 0 if it is not. A return value of -1 indicates an error; in this case, `errno` is set to `EBADF`, indicating an invalid file handle.

## See Also

clearerr, feof, ferror, perror

## Example

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

int fh, count;
char buf[10];

main()
{
    int total = 0;

    fh = open("data", 0_RDONLY);

    /* Cycle until end of file reached: */
    while (!eof(fh))
    {
        /* Attempt to read in 10 bytes: */
        if ((count = read(fh, buf, 10)) == -1)
            perror("Read error");
    }
    close(fh);
}
```
This program opens a file named DATA and reads data from the file until the end of the file is reached. It then uses the function named `eof` to determine when the end of the file was found. If the `read` function reports an error, reading is terminated and the current total is reported.
Summary

```c
#include <process.h>
```

Required only for function declarations

```c
int execl(pathname, arg0, arg1..., argn, NULL);
int execle(pathname, arg0, arg1..., argn, NULL, envp);
int exectl(pathname, arg0, arg1..., argn, NULL);
int execlpe(pathname, arg0, arg1..., argn, NULL, envp);
int execv(pathname, argv);
int execve(pathname, argv, envp);
int execvp(pathname, argv);
int execvpe(pathname, argv, envp);
```

`char *pathname;` Path name of file to be executed
`char *argv[ ];` Array of pointers to arguments
`char *argv[ ];` Array of pointers to environment settings

Description

The `exec` functions load and execute new child processes. When the call is successful, the child process is placed in the memory previously occupied by the calling process. Sufficient memory must be available for loading and executing the child process.

The `pathname` argument specifies the file to be executed 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 file name. If `pathname` does not have a file-name extension or does not end with a period (.), the `exec` functions for the file; if unsuccessful, the extension `.COM` is tried, then `.EXE`. If `pathname` has an extension, only that extension is used. If `pathname` ends with a period, the `exec` calls search for `pathname` with no extension. The `exectl, execle, execvp, and execvpe` routines search for `pathname` (using the same procedures) in the directories specified by
the PATH environment variable.

Arguments are passed to the new process by giving one or more pointers to character strings as arguments in the exec call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the new process must not exceed 128 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are counted.

The argument pointers can be passed as separate arguments (execl, execle, execlp, and execlpe) or as an array of pointers (execv, execve, execvp, and execvpe). At least one argument, arg0, must be passed to the child process (which sees it as argv[0]). Usually, this argument is a copy of the pathname argument. (A different value will not produce an error.) Under versions of DOS earlier than 3.0, the passed value of arg0 is not available for use in the child process. However, under DOS 3.0 and later, the pathname is available as arg0.

The execl, execle, execlp, and execlpe calls are typically used in cases where the number of arguments is known in advance. The argument arg0 is usually a pointer to pathname. The arguments arg1 through argn point to the character strings forming the new argument list. A NULL pointer must follow argn to mark the end of the argument list.

The execv, execve, execvp, and execvpe calls are useful when the number of arguments to the new process is variable. Pointers to the arguments are passed as an array, argv. The argument argv[0] is usually a pointer to pathname. The arguments argv[1] through argv[n] point to the character strings forming the new argument list. The argument argv[n+1] must be a NULL pointer to mark the end of the argument list.

Files that are open when an exec call is made remain open in the new process. In the execl, execlp, execv, and execvpe calls, the child process inherits the environment of the parent. The execle, execlpe, execve, and execvpe calls allow the user to alter the environment for the child process by passing a list of environment settings through the envp argument. The argument envp is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

NAME= value

where NAME is the name of an environment variable and value is the string value to which that variable is set. (Note that value is not enclosed
in double 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.

- **Return Value**

The `exec` functions do not normally return to the calling process. If an `exec` function returns, an error has occurred and the return value is `-1`. The `errno` variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K.</td>
</tr>
<tr>
<td>EACCES</td>
<td>Locking or sharing violation on the specified file (DOS Version 3.0 or later).</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many files open (the specified file must be opened to determine whether it is executable).</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>The specified file is not executable or has an invalid executable file format.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory is available to execute the child process; or the available memory has been corrupted; or an invalid block exists, indicating that the parent process was not allocated properly.</td>
</tr>
</tbody>
</table>

- **See Also**

`abort, exit, _exit, onexit, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, system`

---

**Note**

The `exec` calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the `setmode` routine should be used to set the translation mode of these files to the desired mode.


**excl – execvpe**

Signal settings are not preserved in child processes created by calls to **exec** routines. The signal settings are reset to the default in the child process.

#### Example

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

char *my_env[] = {
    "THIS=environment will be",
    "PASSED=to child.exe by the",
    "EXECLE=and",
    "EXECLPE=and",
    "EXECVE=and",
    "EXECVPE=functions",
    NULL
};

main(argc, argv)
int argc;
char *argv[];
{
    char *args[4];
    int result;

    args[0] = "child";    /* Set up parameters to send */
    args[1] = "execv??";
    args[2] = "two";
    args[3] = NULL;

    switch (argv[1][0])    /* Based on first letter of argument */
    {
        case '1':
            excl("child.exe","child ","execl","two",NULL);
            break;
        case '2':
            execle("child.exe","child","execle","two",NULL,my_env);
            break;
        case '3':
            execlp("child.exe","child","execlp","two",NULL);
            break;
        case '4':
            execlpe("child.exe","child","execlpe","two",NULL,my_env);
            break;
        case '5':
            execv("child.exe",args);
    }
```
break;
case '6':
    execve("child.exe", args, my_env);
    break;
case '7':
    execvp("child.exe", args);
    break;
case '8':
    execvpe("child.exe", args, my_env);
    break;
default:
    printf("Enter a number from 1 to 8 as a \ncommand line parameter.");
    exit();
}
printf("Process was not spawned.0");
printf("Program 'child' was not found.");
}

This program accepts a number in the range 1 through 8 from the command line. Based on the number it receives, it executes one of the eight different procedures that spawn the process named child. For some of these procedures, the CHILD.EXE file must be in the same directory; for others, it only has to be in the same path.
exit — _exit

• Summary

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

void exit(status); Terminate after closing files
void _exit(status); Terminate without flushing stream buffers

int status; Exit status

• Description

The exit and _exit functions terminate the calling process. The exit function first calls the functions registered by atexit, then flushes all buffers and closes all open files before terminating the process. The _exit function terminates the process without processing atexit functions or flushing stream buffers. The status value is typically set to 0 to indicate a normal exit and set to some other value to indicate an error.

Although the exit and _exit calls do not return a value, the low-order byte of status is made available to the waiting parent process, if there is one, after the calling process exits. The status value is available to the DOS command IFERRORLEVEL (see the "MS-DOS User’s Guide").

• Return Value

There is no return value.

• See Also

abort, execl, execle, execlp, execv, execve, execvp, onexit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, system
Example

main()
{
    stream = fopen("data", "w+".);
    printf("About to exit...\nFlush buffers for the file 'data'? (y/n): ");
    aChar = getch();
    aChar = toupper(aChar);
    fprintf(stream, "This will appear in "data" only if \nbuffers are flushed.\n");
    if (aChar == 'Y')
    {
        printf("\nExiting and flushing buffers");
        exit(0);
    }
    else
    {
        printf("\nExiting, but buffers are not flushed");
        _exit(0);
    }
}

This program opens the file named DATA, then prompts the user to choose how to close the file. Based on the user's choice, the program closes the file using the exit function, which flushes the buffers, or the _exit function, which does not.
exp

- Summary

```c
#include <math.h>

double exp(x);

double x; Floating-point value
```

- Description

The `exp` function returns the exponential function of its floating-point argument `x`.

- Return Value

The `exp` function returns $e^x$. On overflow, the function returns `HUGE` and sets `errno` to `ERANGE`; on underflow, `exp` returns 0, but does not set `errno`.

- See Also

`log`

- Example

```c
#include <math.h>

main()
{
    double x, y;
    x = 2.302585093;
    y = exp(x); /* y = 40 */
    printf("The exp(\%f) = \%f", x, y);
}
```

This program displays the value of $e^{2.302585093}$. 

178
-expand

Summary

```c
#include <malloc.h>

char *-_expand(ptr, size);
```

Required only for function declarations

- Description

The `_expand` function changes the size of a previously allocated memory block by attempting to expand or contract the block without moving its location in the heap. The `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 will remain free after the call to `_expand`.

- Return Value

The `_expand` function returns a `char` pointer to the reallocated memory block. Unlike `realloc`, `_expand` cannot move a block to change its size. This means the `ptr` argument to `_expand` is the same as the return value if there is sufficient memory available to expand the block without moving it.

The return value is `NULL` if there is insufficient memory available to expand the block to the given size without moving it. In this case, the item pointed to by `ptr` will have been expanded as much as possible in its current location.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. The new size of the item can be checked with the `msize` function. To get a pointer to a type other than `char`, use a type cast on the return value.
_expand

- See Also

calloc, free, halloc, malloc, _msize, realloc

- Example

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

main()
{
    long *oldptr;
    unsigned int newsize = 64000;

    /* Get original memory: */
    oldptr = (long *)malloc(10000*sizeof(long));
    printf("Size of memory block pointed to by oldPtr = %u\n",
            _msize(oldptr));

    /* Test whether _expand succeeded: */
    if (_expand(oldptr, newsize) != NULL)
        printf("Expand was able to increase block to %u\n",
                _msize(oldptr));
    else
        printf("Expand was able to increase block to only %u\n",
                _msize(oldptr));
}
```

Sample output:

Size of memory block pointed to by oldPtr = 40000
expand was able to increase block to only 44718

This program allocates a block of memory for oldptr and uses _msize to display the size of that block. Next, it uses expand to expand the amount of memory used by oldptr. Finally, it calls _msize again to display the new amount of memory allocated to oldptr.
fabs

- **Summary**

```c
#include <math.h>

double fabs(x);

double x;          Floating-point value
```

- **Description**

The `fabs` function returns the absolute value of its floating-point argument.

- **Return Value**

The `fabs` function returns the absolute value of its argument. There is no error return.

- **See Also**

`abs, cabs, labs`

- **Example**

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

main()
{
    double x,y;

    x = -3.141593;
    y = fabs(x);  /* y = 3.141593 */
    printf("The fabs(\%f) is \%f", x,y);
}
```

This program displays the absolute value of -3.141593.
Summary

#include <stdio.h>

int fclose(stream);  // Close an open stream
FILE *stream;        // Pointer to file structure

int fcloseall();     // Close all open streams

Description

The fclose and fcloseall functions close a stream or streams. All buffers associated with the stream(s) are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned using setbuf are not automatically released.

The fclose function closes the given stream. The fcloseall function closes all open streams except stdin, stdout, stderr, stdaux, and stdprn.

Return Value

The fclose function returns 0 if the stream is successfully closed. The fcloseall function returns the total number of streams closed. Both functions return EOF to indicate an error.

See Also

close, fdopen, fflush, fopen, freopen

183
fclose – fcloseall

Example

```c
#include <stdio.h>

FILE *stream, *stream2;

main()
{
  int numclosed;

  /* Two files are opened: */
  stream = fopen("data", "r");
  stream2 = fopen("data2","w+");

  if (stream == NULL)
    printf("The file 'data' was not opened\n");
  else
  {
    fclose(stream);
    printf("The file 'data' closed\n");
  }

  /* All other files are closed: */
  numclosed = fcloseall();

  printf("The function fcloseall closed %u files\n", numclosed);  
}
```

This program opens files named DATA and DATA2. It uses fclose to close DATA and fcloseall to close all remaining files.
# Summary

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

```
char *fcvt(value, ndec, decptr, signptr);
```

- **Description**

The `fcvt` function converts a floating-point number to a character string. The `value` is the floating-point number to be converted. The `fcvt` function stores the digits of `value` as a string and appends a null character (`'\0'`). The argument `ndec` 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`, the correct digit is rounded according to the FORTRAN F format. If there are fewer than `ndec` digits of precision, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of `value` can be obtained after the call from `decptr` and `signptr`. The argument `decptr` points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The argument `signptr` points to an integer indicating the sign of `value`. The integer is set to 0 if `value` is positive, and is set to a nonzero number if `value` is negative.

- **Return Value**

The `fcvt` function returns a pointer to the string of digits. There is no error return.
fcvt

- See Also

atof, atoi, atol, ecvt, gcvt

---

Note

The ecvt and fcvt functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

---

- Example

```c
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

main()
{
    /* buffer to contain "31415926535",
       ** decimal = 1, sign = 0
    */

    buffer = fcvt(3.1415926535, precision, &decimal, &sign);
    printf("buffer= \"%s\", decimal = %d, sign = %d\n", buffer, decimal, sign);
}
```

This program converts the constant 3.1415926535 to a string and sets the pointer *buffer to point to that string.
# Summary

```c
#include <stdio.h>

FILE *fdopen(handle, type);

int handle;  // Handle referring to open file
char *type;  // Type of access permitted
```

## Description

The `fdopen` function associates an input/output stream with the file identified by `handle`, thus allowing a file opened for "low-level" I/O to be buffered and formatted. (See Section 4.7, "Input and Output," for an explanation of stream I/O versus low-level I/O.) The `type` character string specifies the type of access requested for the file, as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open for reading (the file must exist).</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Open an empty file for writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Open for writing at the end of the file (appending); create the file first if it doesn't exist.</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open for both reading and writing (the file must exist).</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Open an empty file for both reading and writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open for reading and appending; create the file first if it doesn't exist.</td>
</tr>
</tbody>
</table>

### Important

Use the "w" and "w+" modes with care, as they can destroy existing files.
fdopen

The specified type must be compatible with the access mode and/or sharing modes with which the file was opened. It is the user’s responsibility to ensure that this compatibility is maintained.

When a file is opened with "a" or "a+" type, all write operations take place at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for “update”). However, when switching from reading to writing or vice versa, there must be an intervening fseek or rewind operation. The current position can be specified for the fseek operation, if desired.

In addition to the values listed above, one of the following characters can be appended to the type string to specify the translation mode for new lines.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Open in text (translated) mode; carriage-return–line-feed combinations (CR-LF) are translated into a single line feed (LF) on input; line-feed characters are translated to carriage-return–line-feed combinations on output.</td>
</tr>
<tr>
<td>b</td>
<td>Open in binary (untranslated) mode; the above translations are suppressed.</td>
</tr>
</tbody>
</table>

If t or b is not given in the type string, the translation mode is defined by the default mode variable _fmode.

Return Value

The fdopen function returns a pointer to the open stream. A NULL pointer value indicates an error.
See Also
dup, dup2, fclose, fcloseall, fopen, freopen, open

Example

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

FILE *stream;
int fh;

main()
{
  fh = open("data", O_RDONLY);

  /* Buffer associated with "fh": */
  stream = fdopen(fh,"r");
  if (stream == NULL)
    printf( "Error in fdopen attempt.\n" );
  else
    printf( "Input buffer successfully associated with 'data'" );
}
```

This program opens a file named DATA and uses **fdopen** to associate an input stream with DATA.
feof

■ Summary

#include <stdio.h>

int feof(stream);
FILE *stream; Pointer to file structure

■ Description

The feof function determines whether the end of the given stream has been reached. Once end-of-file is reached, read operations return an end-of-file indicator until the stream is closed or rewind is called.

■ Return Value

The feof function returns a non-zero value after the first read operation which attempts to read past the end of the file. It returns 0 if the current position is not end-of-file. There is no error return.

■ See Also

clearerr, eof, ferror, perror

Note

The feof function is implemented as a macro.

■ Example

#include <stdio.h>

#define BUF_SIZE 100

char string[BUF_SIZE];
FILE *stream;
main()
{
    stream = fopen("data", "r");
    /* Process input until eof occurs: */
    while (fgets(string, BUF_SIZE, stream))
    {

    190
printf("%s", string);
}

if (feof(stream)) printf("EOF reached\n");
else printf("Error reading stream\n");

This program opens a file named DATA and usesfeof to indicate when the end of DATA has been reached.
ferror

- **Summary**

```c
#include <stdio.h>

int ferror(stream);
FILE *stream; Pointer to file structure
```

- **Description**

The `ferror` function tests for a reading or writing error on the given `stream`. If an error has occurred, the error indicator for the `stream` remains set until the stream is closed or rewound or until `clearerr` is called.

- **Return Value**

The `ferror` function returns a nonzero value to indicate an error on the given `stream`. The return value 0 means no error has occurred.

- **See Also**

`clearerr, eof, feof, fopen, perror`

---

**Note**

The `ferror` function is implemented as a macro.
Example

```c
#include <stdio.h>

FILE *stream;
char *string = "This should never be written";

main()
{
    /* Note that "r" assures an error: */
    stream = fopen("data", "r");

    fprintf(stream,"%s\n",string);
    if (ferror(stream))
    {
        fprintf(stderr,"Write error\n");
        clearerr(stream);
    }
}
```

This program opens a file named DATA for reading and tries to write to DATA, causing an error. The program uses **ferror** to detect the error, then clears the error.
fflush

Summary

# include <stdio.h>

int fflush(stream);
FILE *stream; Pointer to file structure

Description

If the specified stream is open for output, fflush causes the contents of the buffer associated with the stream to be written to the associated file. If the stream is open for input, fflush clears the contents of the buffer. The fflush function reverses the effect of any prior call to ungetc against stream.

The stream remains open after the call. The fflush function has no effect on an unbuffered stream.

Return Value

The fflush function returns the value 0 if the buffer was successfully flushed. The value 0 is also returned in cases where the specified stream has no buffer or is open for reading only. A return value of EOF indicates an error.

See Also

fclose, flushall, setbuf

Note

Buffers are automatically flushed when they are full, when the stream is closed, or when a program terminates normally without closing the stream.
Example

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

FILE *stream;
char buffer[BUFSIZ];

main()
{
  int result;

  /* Redirect stdout to "data" */
  stream = freopen("data", "w", stdout);

  printf("This is the output of child:\n\n");

  /* Now make sure printf() output goes to "data" before child's output does: */
  result = fflush(stream);

  spawnl(P_WAIT, "child.exe", "child", "one", "two", NULL);

  printf("--------------------------------------------\n\n");
}
```

This program first redirects `stdout` to a file named `DATA`. It uses `printf` to write to `DATA`, then uses `fflush` to guarantee that the output from `printf` is written before the output from the child process.
_ffree

- Summary

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

void _ffree(ptr);
char far *ptr;  // Pointer to allocated memory block
```

- Description

The _ffree function deallocates a memory block outside the default data segment. The argument `ptr` points to a memory block previously allocated through a call to _fmalloc. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is again available for allocation.

- Return Value

There is no return value.

- See Also

_fmalloc, free, malloc

---

**Note**

Attempting to free an invalid `ptr` (a pointer not allocated with _fmalloc) may affect subsequent allocation and cause errors.
```c
#include <malloc.h>
#include <stdio.h>

char far *alloc;

main()
{
    /* Test for a valid pointer: */
    if ((alloc = _fmalloc(100)) == NULL)
        printf("unable to allocate memory\n");
    else
    {
        printf("memory sucessfully allocated and \nnow will be deallocated\n");

        _ffree(alloc);    /* Free the allocated memory */
        printf("done\n");
    }
}

This program uses _fmalloc to allocate 100 bytes of memory outside the default data segment. They are then deallocated using ffree.
```
fgetc – fgetchar

- Summary

```c
#include <stdio.h>
int fgetc(stream);
FILE *stream;  // Pointer to file structure

int fgetchar();  // Read a character from stdin
```

- Description

The `fgetc` function reads a single character as an `unsigned int` character converted to an `int` from the input `stream` at the current position. The function then increases the associated file pointer (if any) to point to the next character. The `fgetchar` function is equivalent to `fgetc(stdin)`.

- Return Value

The `fgetc` and `fgetchar` functions return the character read. A return value of `EOF` may indicate an error or end-of-file; however, the `EOF` value is also a legitimate integer value, so `feof` or `ferror` should be used to verify an error or end-of-file condition.

- See Also

`fputc, fputchar, getc, getchar`

---

**Note**

The `fgetc` and `fgetchar` routines are identical to `getc` and `getchar`, but are functions, not macros.
Example

```c
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;

main()
{
    /* Open file to read line from: */
    stream = fopen("fgetc.c", "r");

    /* Read in first 80 characters and */
    /* place them in "buffer": */

    ch = fgetc(stream);
    for (i=0; (i < 80) && (feof(stream) == 0) && (ch != 'O'); i++) {
        buffer[i]=ch;
        ch = fgetc(stream);
    }

    buffer[i] = '\0';        /* Add null to end string */
    printf("%s\n", buffer);
}
```

This program uses `getc` to read the first 80 input characters (or until the end of input) and place them into a string named `buffer`. 
fgetpos

- Summary

```c
#include <stdio.h>

int fgetpos(stream, pos);

FILE *stream;       // Target stream
fpos_t *pos;        // Position indicator storage
```

- Description

The `fgetpos` function gets the current value of `stream`'s file position indicator and stores it in the object that `pos` points to. The `fsetpos` function can later use information stored in `pos` to reset `stream`'s pointer to its position at the time `fgetpos` was called.

- Return Value

If successful, the `fgetpos` function returns zero. On failure, it returns a nonzero value and sets `errno` to a nonzero value.

- See Also

`fsetpos`

- Example

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

FILE *stream;
fpos_t position;
char list[100];

main()
{
    stream = fopen("data","rb");
    /* Move the pointer by reading data: */
    fread(list,sizeof(char),100, stream);
    /* Get position after read: */
```
fgetpos

fgetpos(stream, position)
printf("position = %ld\n", position);
}

This program opens a file named DATA for reading and tries to read 100 characters. It then uses fgetpos to determine the position of the file pointer and displays this position.
fgets

- **Summary**

```c
#include <stdio.h>

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

- **Description**

The `fgets` function reads a string from the input `stream` and stores it in `string`. Characters are read from the current `stream` position up to and including the first new-line character (`'\n'`), up to the end of the stream, or until the number of characters read is equal to `n - 1`, whichever comes first. The result is stored in `string`, and a null character (`'\0'`) is appended. The new line, if read, is included in the `string`. If `n` is equal to 1, `string` is empty (""").

The `fgets` function is similar to the library function `gets`; however, `gets` replaces the new-line character with the null character.

- **Return Value**

If successful, the `fgets` function returns `string`. A NULL return value indicates an error or end-of-file condition. Use `feof` or `ferror` to determine whether the NULL value represents an error or end-of-file.

- **See Also**

`fputs`, `gets`, `puts`
Example

#include <stdio.h>

FILE *stream;
char line[100], *result;

main()
{
    stream = fopen("fgets.c", "r");
    /* "line" will point to the string: */
    result = fgets(line, 100, stream);

    printf("%s", line);
}

This program uses fgets to display a line from a file on the screen. Note that fgets does not remove the new-line character (\n) from each line.
_fheapchk

- Summary

#include <malloc.h>

int _fheapchk();

- Description

The _fheapchk routine performs a minimal consistency check on the far heap. The consistency check determines whether all the heap entries are within the bounds of the heap's current memory allocation.

- Return Value

The function returns a 0 if successful or a non-zero value otherwise, corresponding to one of the following manifest constants (defined in malloc.h):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_HEAPEMPTY</td>
<td>The far heap is not initialized.</td>
</tr>
<tr>
<td>_HEAPBADBEGIN</td>
<td>Couldn’t find the initial header information.</td>
</tr>
<tr>
<td>_HEAPBADNODE</td>
<td>Found a bad node, or the far heap is damaged.</td>
</tr>
</tbody>
</table>

- See Also

_fheapset, _fheapwalk, _nheapchk, _nheapset, _nheapwalk

- Example
# _fheapset

- **Summary**

```c
#include <malloc.h>

int _fheapset(fill);
```

int fill;  // Fill character

- **Description**

The `_fheapset` routine sets the contents of the far heap free nodes to the specified fill character.

- **Return Values**

The function returns 0 if successful or a non-zero value otherwise.

- **See Also**

- **Example**
_fheapwalk

- Summary

```c
#include <malloc.h>

int _fheapwalk(&farentry);
```

```c
_heapinfo *farentry;  // Far heap entry structure
```

- Description

The _fheapwalk routine returns the address of the next far heap entry structure. The structure contains a pointer to the heap entry, size of the heap entry, and a free/in-use flag. The structure is defined in malloc.h.

- Return Value

- See Also

- Example
fieetomsbin – fmsbintoeiee

■ Summary

#include <math.h>

int fieetomsbin(src4, dst4);        // IEEE floating-point to MS binary floating-point
int fmsbintoeiee(src4, dst4);       // MS binary floating-point to IEEE floating-point
float *src4, *dst4;

■ Description

The fieetomsbin routine converts a single-precision floating-point number in IEEE format to Microsoft binary format. The fmsbintoeiee routine converts a floating-point number in Microsoft binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random access data files created with Microsoft BASIC (which store floating-point numbers in the Microsoft binary format), and vice versa.

The argument src4 points to the float value to be converted. The result is stored at the location given by dst4.

■ Return Value

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

■ See Also

dieetomsbin, dmsbintoeiee

■ Note

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

- **Summary**

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

long filelength(handle);
int handle;     // Handle referring to open file
```

- **Description**

The `filelength` function returns the length, in bytes, of the file associated with the given `handle`.

- **Return Value**

The `filelength` function returns the file length in bytes. A return value of -1L indicates an error, and `errno` is set to `EBADF` to indicate an invalid file handle.

- **See Also**

`chsize`, `fileno`, `fstat`, `stat`

- **Example**

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

FILE *stream;    // stream
long length;     // length;

main()
{
    stream = fopen("data","r");

    /* Get length or -1L if function fails: */
    length = filelength(fileno(stream));

    if (length == -1L)    /* If function failed... */
        printf("filelength failed");
    else
        printf( "file length is %ld\n", length );
}
```
This program opens a file named DATA and uses filelength to determine its length. If filelength fails, it returns -1L and the program displays a message indicating the failure. Otherwise, the program displays the length of DATA.
fileno

**Summary**

```c
#include <stdio.h>

int fileno(stream);
```

FILE *stream;  // Pointer to file structure

**Description**

The `fileno` function returns the file handle currently associated with the given `stream`. If more than one handle is associated with the stream, the return value is the handle assigned when the stream was initially opened.

**Return Value**

The `fileno` function returns the file handle. There is no error return. The result is undefined if `stream` does not specify an open file.

**See Also**

`fdopen`, `filelength`, `fopen`, `freopen`

---

**Note**

The `fileno` routine is implemented as a macro.

---

**Example**

```c
#include <stdio.h>

main()
{
    int result;

    result = fileno(stderr);       // result is 2
    printf("The file handle for stderr is %d\n", result);
}
```
This program uses `fileno` to obtain the file handle of `stderr`. 
Summary

#include <math.h>

double floor(x);
double x; Floating-point value

Description

The floor function returns a floating-point value representing the largest integer that is less than or equal to x.

Return Value

The floor function returns the floating-point result. There is no error return.

See Also

ceil, fmod

Example

#include <math.h>

main()
{
    double y;
    y = floor(2.8); /* y is 2.0 */
    printf("The floor of 2.8 is %f\n",y);
    
    y = floor(-2.8); /* y is -3.0 */
    printf("The floor of -2.8 is %f\n",y);
}

This example displays the largest integers less than or equal to the floating-point values 2.8 and -2.8.
flushall

■ Summary

#include <stdio.h>

int flushall();

■ Description

The flushall function causes the contents of all buffers associated with open output streams to be written to the associated files. All buffers associated with open input streams are cleared of their current contents; the next read operation (if there is one) then reads new data from the input files into the buffers.

All streams remain open after the call to flushall.

■ Return Value

The flushall function returns the number of open streams (input and output). There is no error return.

■ See Also

fflush

Note

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.
Example

```c
#include <stdio.h>

main()
{
    int numflushed;

    numflushed = flushall();
    printf("There were %d streams flushed\n", numflushed);
}
```

This program uses `flushall` to flush all buffers, including `stdin`, `stdout`, and `stderr`, and prints the number of open streams.
-fmalloc

- Summary

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

char far *-fmalloc(size);
unsigned size;  Bytes in allocated block

- Description

The _fmalloc function allocates a memory block of at least size bytes outside the default data segment. (The block may be larger than size bytes, due to space required for alignment.)

- Return Value

The _fmalloc function returns a far pointer to an char. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than char, use a type cast on the return value.

If sufficient memory is not available outside the default data segment, the allocation will be retried using the default data segment. If there is still insufficient memory available, the return value is NULL.

- See Also

_ffree, _fmsize, malloc, realloc

- Example

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

int far *intarray;

main()
{
    /* Test for a valid pointer: */
    if ((intarray = (int far *)_fmalloc(20*sizeof(int))) == NULL)
        printf("Unable to allocate memory\n");
else
    printf("Memory successfully allocated");
}

This program uses `_fmalloc` to allocate a block of memory large enough for 20 integers outside the default data segment.
fmod

- Summary

```c
#include <math.h>

double fmod(x, y);

double x; Floating-point values

double y;
```

- Description

The `fmod` function calculates the floating-point remainder \( f \) of \( x/y \), such that \( x = iy + f \), where \( i \) is an integer, \( f \) has the same sign as \( x \), and the absolute value of \( f \) is less than the absolute value of \( y \).

- Return Value

The `fmod` function returns the floating-point remainder. If \( y \) is 0, the function returns 0.

- See Also

`ceil`, `fabs`, `floor`

- Example

```c
#include <math.h>

main()
{
    double x, y, z;

    x = -10.0;
    y = 3.0;
    z = fmod(x, y); /**< z is -1.0 */
    printf("fmod(%.2f, %.2f) is %f", x, y, z);
}
```

This program displays the floating-point remainder of \(-10/3\).
- **Summary**

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

```c
unsigned _fmsize(ptr);
```

```c
char far *ptr;  
Pointer to memory block
```

- **Description**

The `_fmsize` function returns the size in bytes of the memory block allocated by a call to `_fmalloc`.

- **Return Value**

The `_fmsize` function returns the size in bytes as an unsigned integer.

- **See Also**

`_ffree, _fmalloc, malloc, _msize, _nfree, _nmalloc, _nmsize`

- **Example**

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

main()
{
    char far *stringarray;

    stringarray = _fmalloc(200*sizeof(char));
    if (stringarray != NULL) /* "fmalloc" succeeded */
        printf("%u bytes allocated\n", _fmsize(stringarray));
    else /* "fmalloc" failed */
        printf("Allocation request failed.\n");
}
```

This program uses `_fmsize` to determine the size, in bytes, of the allocated block of memory, then displays this size.
fopen

■ Summary

#include <stdio.h>

FILE *fopen(pathname, type);
char *pathname; Path name of file
char *type; Type of access permitted

■ Description

The fopen function opens the file specified by pathname. The character string type specifies the type of access requested for the file, as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open for reading. If r is the first character in the type string, and the file does not exist or cannot be found, the fopen call will fail.</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Open an empty file for writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Open for writing at the end of the file (appending); create the file first if it doesn’t exist.</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open for both reading and writing (the file must exist).</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Open an empty file for both reading and writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open for reading and appending; create the file first if it doesn’t exist.</td>
</tr>
</tbody>
</table>

Note

Use the "w" and "w+" types with care, as they can destroy existing files.

When a file is opened with the "a" or "a+" type, all write operations occur at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data
fopen

cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching between reading and writing, there must be an intervening fseek or rewind operation. The current position can be specified for the fseek operation, if desired.

In addition to the values listed above, one of the following characters can be appended to the type string to specify the translation mode for newlines:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Open in text (translated) mode. In this mode, carriage-return–line-feed combinations (CR-LF) are translated into a single line feed (LF) on input, line-feed characters are translated to carriage-return–line-feed combinations on output. Also, CONTROL-Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, the run-time checks for a CONTROL-Z at the end of the file and removes it, if possible. This is because using the fseek and ftell functions to move within a file that ends with a CONTROL-Z may cause fseek to behave improperly near the end of the file. Notice that t is not part of the ANSI standard for open, but is instead an extension.</td>
</tr>
<tr>
<td>b</td>
<td>Open in binary (untranslated) mode; the above translations are suppressed.</td>
</tr>
</tbody>
</table>

If t or b is not given in the type string, the translation mode is defined by the default mode variable _fmode.

■ Return Value

The fopen function returns a pointer to the open file. A NULL pointer value indicates an error.
fopen

- See Also
fclose, fcloseall, fdopen, ferror, fileno, freopen, open, setmode

- Example

```c
#include <stdio.h>

FILE *stream;

main() {
    /* Attempt to open the file: */
    if ((stream = fopen("data","r")) == NULL)
        printf("Could not open file\n");
    else
        printf( "File opened for reading\n" );
}
```

Sample command line:

```
update employ.dat
```

Output:

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

This program uses `fopen` to open a file named DATA for input.
FP_OFF – FP_SEG

Summary

#include <dos.h>

unsigned FP_OFF(longptr);
unsigned FP_SEG(longptr);
char far *longptr

Description

The FP_OFF and FP_SEG macros can be used to set or get the offset and segment, respectively, of the long pointer longptr.

Return Value

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.

See Also

segread

Example

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

char far *p;
unsigned int seg_val;
unsigned int off_val;

main()
{
p = _fmalloc(100);    /* Point pointer at something */

seg_val = FP_SEG(p); /* Get address pointed to */
off_val = FP_OFF(p);
printf("Segment is %d; Offset is %d\n", seg_val, off_val);
This program uses **FP_SEG** and **FP_OFF** to obtain the segment and offset of the long pointer \( p \).
# Summary

```c
#include <float.h>

void _fpreset();
```

Reinitialize floating-point math package

## Description

The `_fpreset` function reinitializes the floating-point math package. This function is usually used in conjunction with `signal`, `system`, or the `exec` or `spawn` family of routines.

If a program traps floating-point error signals (SIGFPE) with `signal`, it can safely recover from floating-point errors by invoking `_fpreset` and doing a `longjmp`.

---

**Note**

On DOS versions prior to 3.0, a child process executed by `exec`, `spawn`, or `system` might affect the floating-point state of the parent process if an 8087 or 80287 coprocessor is used. Therefore, if you are using either an 8087 or an 80287, the following precautions are recommended:

- The `exec`, `spawn`, and `system` functions should not be called during the evaluation of a floating-point expression.

- The `_fpreset` function should be called after these routines if there is a possibility of the child process performing any floating-point operations using an 8087 or 80287.

---

## Return Value

There is no return value.
_fpreset

- See Also

`exec`, `execle`, `exep`, `execve`, `execvp`, `execvpe`, `signal`, `spawn`, `spawnle`, `spawnlp`, `spawlpe`, `spawnv`, `spawne`, `spawnp`, `spawvpe`

- Example

```c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <float.h>

int fphandler();
jmp_buf mark;
double a = 1.0, b = 0.0, c;

main()
{
    /* Set up pointer to error handler: */
    if (signal(SIGFPE, fphandler) != (int(*)(())) -1)
        abort();
    if (setjmp(mark) == 0)        /* Save stack environment */
    {
        /* Generate divide by zero error: */
        c = a/b;
        printf("Should never get here\n");
    }
    printf("Recovered from floating-point error\n");
}

int fphandler(sig, num)
int sig, num;
{
    printf("signal = %d subcode = %d\n", sig, num);
    /* Initialize floating-point package: */
    _fpreset();

    /* Restore environment; return -1: */
    longjump(mark, -1)
}
```

224
This program uses `signal` to set up a routine for handling floating-point errors. This routine, `fphandler`, displays an error message and reinitializes the floating-point math package using `_fpreset`. 
fprintf

Summary

#include <stdio.h>

int fprintf(stream, format-string[], argument...);

FILE *stream;  // Pointer to file structure
const char *format-string;  // Format control string

Description

The fprintf function formats and prints a series of characters and values to the output stream. Each argument (if any) is converted and output according to the corresponding format specification in 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.

Return Value

The fprintf function returns the number of characters printed.

See Also

cprintf, fscanf, printf, sprintf
Example

#include <stdio.h>
#include <process.h>

FILE *stream;
int i = 10;
double fp = 1.5;
char *s = "this is a string";
char c = '\n';

main()
{
    stream = fopen("results", "w");

    /* Print "this is a string" followed */
    /* by a new line: */
    fprintf(stream, "%s%c", s, c);

    /* Print 10 followed by a new line: */
    fprintf(stream, "%d\n", i);

    fprintf(stream, "%f\n", fp);       /* Print 1.500000 */

    fclose(stream);

    /* Display the results on the screen: */
    system("type results");
}

This program uses `fprintf` to format various data and print them to the file named RESULTS. It then displays results on the screen.
fputc – fputchar

- Summary

```c
#include <stdio.h>

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

- Description

The `fputc` function writes the single character `c` to the output `stream` at the current position. The `fputchar` function is equivalent to `fputc(c, stdout)`.

- Return Value

The `fputc` and `fputchar` functions return the character written. A return value of EOF may indicate an error; however, since the EOF value is also a legitimate integer value, use `ferror` to verify an error condition.

---

**Note**

The `fputc` and `fputchar` routines are identical to `putc` and `putchar`, but are functions, not macros.

---

- See Also

`fgetc`, `fgetchar`, `putc`, `putchar`
Example

```c
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;

main()
{
    stream = stdout;

    /* Demonstrate "fputc"; */
    /* Set up buffer: */
    /*
    strcpy(buffer, "This is a test of fputc!!\n");
    /* Print line to stream */
    for (i = 0; ( i < 81 ) &&
         ((ch = fputc(buffer[i], stream)) != EOF); i++) ;
    */

    /* Demonstrate "fputchar"; */
    /* Set up buffer: */
    /*
    strcpy(buffer, "This is a test of fputchar!!");
    /* Print line to stream */
    for (i = 0; ( i < 81 ) &&
         ((ch = fputchar(buffer[i])) != EOF); i++) ;
}
```

This program uses `fputc` and `fputchar` to send a character array to `stdout`. 
fputs

• Summary

#include <stdio.h>

int fputs(string, stream); Write a string to stream
char *string; String to be output
FILE *stream; Pointer to file structure

• Description

The fputs function copies string to the output stream at the current position. The terminating null character ('\0') is not copied.

• Return Value

The fputs function returns the last character output. If the input string is empty, the return value is 0. The return value EOF indicates an error.

• See Also

fgets, gets, puts

• Example

#include <stdio.h>
FILE *stream;
main()
{
    int result;
    stream = stdout;
    result = fputs("Data files have been updated\n", stream);
}

This program uses fputs to write a single line to a stream.
# Summary

#include <stdio.h>

size_t fread(buffer, size, count, stream);

void *buffer;                           // Storage location for data
size_t size;                           // Item size in bytes
size_t count;                          // Maximum number of items to be read
FILE *stream;                          // Pointer to file structure

## Description

The **fread** function reads as many as **count** items of length **size** from the input **stream** and stores them in the given **buffer**. The file pointer associated with **stream** (if there is one) is increased by the number of bytes actually read.

If the given **stream** was opened in text mode, carriage-return–line-feed pairs (CR-LF) are replaced with single line-feed characters (LF). The replacement has no effect on the file pointer or the return value.

If an error occurs, the resulting file pointer position is indeterminate. If a partial item is read, its value is indeterminate.

## Return Value

The **fread** function returns the number of full items actually read, which may be less than **count** if an error occurs or the file end is encountered before reaching **count**.

The **feof** or **ferror** function should be used to distinguish a read error from an end-of-file condition. If **size** or **count** is zero, **fread** returns zero and the buffer contents are unchanged.

## See Also

fwrite, read
fread

- Example

```c
#include <stdio.h>

FILE *stream;
long list[100];
int numread;
int numwritten;

main()
{
    /* Open file in "binary" mode: */
    if ((stream = fopen("data", "w+b")) != NULL)
    {
        /* Write 100 long integers to "stream": */
        numwritten = fwrite((char *)list,sizeof(long),100,stream);
        printf("Wrote %d items\n", numwritten);
    }
    else
        printf("Problem opening the file");

    fclose(stream);

    if ((stream = fopen("data", "r+b")) != NULL)
    {
        /* Attempt to read in 100 long ints: */
        numread = fread((char *)list,sizeof(long),100,stream);
        printf("Number of items read = %d\n", numread);
    }
    else
        printf("Was not able to open the file");
}
```

This program opens a file named DATA.BIN and writes 100 long integers to the file. It then tries to open DATA.BIN and read in 100 long integers. If the attempt succeeds, the program displays the number of actual items read.
Summary

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

void free(ptr);
char *ptr;  Pointer to allocated memory block

Description

The free function deallocates a memory block. The argument ptr points to a memory block previously allocated through a call to calloc, malloc, or realloc. The number of bytes freed is the number of bytes specified when the block was allocated (or reallocated, in the case of realloc). After the call, the freed block is available for allocation.

Return Value

There is no return value.

See Also

calloc, malloc, realloc

Note

Attempting to free an invalid ptr (a pointer not allocated with calloc, malloc, or realloc) may affect subsequent allocation and cause errors.
free

- Example

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

char *alloc;

main()
{
    /* If there is nothing to free... */
    if ((alloc = malloc(100)) == NULL)
        printf("Unable to allocate memory");
    else
    {
        /* Free memory for the heap: */
        free(alloc);
        printf("100 bytes freed\n");
    }
}
```

This program uses `malloc` to allocate a block of memory and then uses `free` to free this block.
# Summary

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

Required only for function declarations

```c
unsigned int _freect(size);
unsigned int size;
```

Item size in bytes

## Description

The `_freect` function tells you how much memory is available for dynamic memory allocation by returning the approximate number of times your program can call `malloc` to allocate an item of a given size in the default data segment.

## Return Value

The `_freect` function returns the number of calls as an unsigned integer.

## See Also

`calloc`, `_expand`, `malloc`, `_memavl`, `_msize`, `realloc`
# Example

```c
#include <malloc.h>

main()
{
    int i;

    /* First report on the free space: */
    printf("Approximate # of times program can call malloc\n");
    printf("to allocate a single integer = %u\n",
            _freect(sizeof(int)));

    /* Allocate space for 1000 integers: */
    for (i = 0; i < 1000; ++i)
        malloc(sizeof(int));

    /* Report again on the free space: */
    printf("Approximate # of times program can call malloc\n");
    printf("to allocate a single integer = %u\n",
            _freect(sizeof(int)));
}
```

Sample output:

```
Approximate # of times program can call malloc
to allocate a single integer = 15268

Approximate # of times program can call malloc
to allocate a single integer = 14266
```

This program determines how much free space is available for integers in the default data segment. Then it allocates space for 1000 integers and checks the space again using `__freect`.  

236
Summary

#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

Description

The freopen function closes the file currently associated with stream and reassigns stream to the file specified by pathname. The freopen function is typically used to redirect the preopened files stdin, stdout, stderr, stdaux, and stdprn to files specified by the user. The new file associated with stream is opened with the given type, which is a character string specifying the type of access requested for the file, as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open for reading. If r is the first character in the type string and the file does not exist or cannot be found, the freopen call will fail.</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Open an empty file for writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Open for writing at the end of the file (appending); create the file first if it doesn't exist.</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open for both reading and writing (the file must exist).</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Open an empty file for both reading and writing; if the given file exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open for reading and appending; create the file first if it doesn't exist.</td>
</tr>
</tbody>
</table>

Note

Use the "w" and "w+" types with care, as they can destroy existing files.
**freopen**

When a file is opened with the "a" or "a+" types, all write operations take place at the end of the file. Although the file pointer can be repositioned using `fseek` or `rewind`, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for “update”). However, when switching between reading and writing, there must be an intervening `fseek` or `rewind` operation. The current position can be specified for the `fseek` operation, if desired.

In addition to the values listed above, one of the following characters may be appended to the type string to specify the translation mode for new lines:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Open in text (translated) mode; carriage-return-line-feed combinations are translated into a single line feed on input; line-feed characters are translated to carriage-return-line-feed combinations on output. Note that the t option is not part of the ANSI standard for <code>freopen</code>, but is instead an extension.</td>
</tr>
<tr>
<td>b</td>
<td>Open in binary (untranslated) mode; the above translations are suppressed.</td>
</tr>
</tbody>
</table>

If t or b is not given in the type string, the translation mode is defined by the default mode variable `_fmode`.

**Return Value**

The `freopen` function returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a NULL pointer value.

**See Also**

`fclose`, `fcloseall`, `fdopen`, `fileno`, `fopen`, `open`, `setmode`
Example

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

FILE *stream, *errstream;

main()
{
    /* Reassign "stdout" to "data2": */
    stream = freopen("data2", "w+", stdout);

    /* If reassignment failed: */
    if (stream == NULL)
        fprintf(stderr, "error on freopen\n");
    else
    {
        fseek(stream, OL, SEEK_END);
        fprintf(stream,"This will go to the file 'data2'\n");
        fprintf("'stdout' successfully reassigned\n");
        system("type data2");
    }
}
```

This program reassigns `stdout` to the file named `DATA2` and writes a line to that file.
frexp

- Summary

#include <math.h>

double frexp(x, expptr);
double x;
int *expptr;
Floating-point value
Pointer to stored integer exponent

- Description

The frexp function breaks down the floating-point value x into a mantissa m and an exponent n such that the absolute value of m is greater than or equal to 0.5 and less than 1.0 and \( x = m \times 2^n \). The integer exponent n is stored at the location pointed to by expptr.

- Return Value

The frexp function returns the mantissa m. If x is 0, the function returns 0 for both the mantissa and exponent. There is no error return.

- See Also

ldexp, modf

- Example

#include <math.h>

main()
{
    double x, y;
    int n;

    x = 16.4;
    y = frexp(x, &n); /* y is .5125 and n is 5 */

    printf("y = %f and n = %d", y, n);
}

240
This program calculates \texttt{frexp (16.4, &n)}, then displays \( y \) and \( n \).
Summary

#include <stdio.h>

int fscanf(stream, format-string[, argument...]);
FILE *stream; Pointer to file structure
const char *format-string; Format-control string

Description

The fscanf function reads data from the current position of the specified stream into the locations given by arguments (if any). Each argument must be a pointer to a variable with a type that corresponds to a type specifier in 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.

Return Value

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

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

See Also

cscanf, fprintf, scanf, sscanf
fscanf

- Example

```c
#include <stdio.h>

FILE *stream;
long l;
float fp;
char s[81];
char c;

int result;

main()
{
    stream = fopen("data", "w+");
    /* Write data to the file: */
    fprintf(stream,"%s %ld %f%c%c","a-string",
            65000, 3.14159, 'x', EOF);
    
    /* Set pointer to beginning of file: */
    fseek(stream,0,SEEK_SET);
    
    /* Read data back from file: */
    result = fscanf(stream, "%s", &s);
    result = fscanf(stream, "%ld", &l);
    result = fscanf(stream, "%f", &fp);
    result = fscanf(stream, "%c", &c);
    
    /* Output data read: */
    printf("%s\n", s);
    printf("%ld\n", l);
    printf("%f\n", fp);
    printf("%c\n", c);
}
```

This program first opens a file named DATA. It then uses `fscanf` to accept various types of input data and `printf` to display these data on the screen.
Summary

#include <stdio.h>

int fseek(stream, offset, origin);

FILE *stream;
long offset;
int origin;

Description

The `fseek` function moves the file pointer (if any) 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 read or a write.

The argument `origin` must be one of the following constants defined in `stdio.h`:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>Beginning of file</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Current position of file pointer</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>End of file</td>
</tr>
</tbody>
</table>

The `fseek` function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However, an attempt to position the pointer in front of the beginning of the file causes an error.

The `fseek` function clears the end-of-file indicator and reverses the effect of any prior calls to `ungetc` against `stream`.

Return Value

The `fseek` function returns the value 0 if the pointer was successfully moved. A nonzero return value indicates an error. On devices incapable of seeking (such as terminals and printers), the return value is undefined.
fseek

- See Also

ftell, lseek, rewind

---

Note

For streams opened in text mode, fseek has limited use because carriage-return–line-feed translations can cause fseek to produce unexpected results. The only fseek operations guaranteed to work on streams opened in text mode are the following:

- seeking with an offset of 0 relative to any of the origin values
- seeking from the beginning of the file with an offset value returned from a call to ftell

---

- Example

```c
#include <stdio.h>

FILE *stream;

main()
{
    char line[81];
    int result;

    stream = fopen("data","w+");
    fprintf(stream,"This is the first line in file 'data'.\n");

    result = fseek(stream,0L,SEEK_SET); /* Attempt to */
          /* position pointer */
    if (result)
        perror("Fseek failed");
    else
    {
        printf("File pointer is set to the beginning of file.\n");
        fgets(line,80,stream);
        printf("%s",line);
    }
}
```

244
This program opens a file named DATA, then repositions the file pointer to the beginning of the file.
fsetpos

- **Summary**

```c
#include <stdio.h>

int fsetpos(stream, pos);
FILE *stream;          // Target stream
const fpos_t *pos;     // Position indicator storage
```

- **Description**

The `fsetpos` function sets the file position indicator for `stream` to the value of `pos`, which is obtained in a prior call to `fgetpos` against `stream`.

The function clears the end-of-file indicator and undoes any effects of the `ungetc` function on `stream`. After calling `fsetpos`, the next operation on `stream` may be either input or output.

- **Return Value**

If successful, the `fsetpos` function returns zero. On failure, the function returns a nonzero value and sets `errno` to a nonzero value.

- **See Also**

`fgetpos`

- **Example**

```c
#include <stdio.h>
FILE *stream;

main()
{
    const fpos_t *pos;
    stream = fopen("data","w+");
    /* Attempt to position pointer: */
    if (fsetpos(stream, pos) != 0) perror("fsetpos failed");
}
```
This program opens a file named DATA, then repositions the file pointer to the beginning of the file.
fstat

■ Summary

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

int fstat(handle, buffer);
int handle;
struct stat *buffer;

■ Description

The fstat function obtains information about the open file associated with the given handle and stores it in the structure pointed to by buffer. The structure, whose type stat is defined in sys/stat.h, contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_mode</td>
<td>Bit mask for file-mode information. S_IFCHR bit set if handle refers to a device. S_IFREG bit set if handle refers to an ordinary file. User read/write bits set according to the file's permission mode.</td>
</tr>
<tr>
<td>st_dev</td>
<td>Either the drive number of the disk containing the file, or handle in the case of a device (same as st_rdev).</td>
</tr>
<tr>
<td>st_rdev</td>
<td>Either the drive number of the disk containing the file, or handle in the case of a device (same as st_dev).</td>
</tr>
<tr>
<td>st_nlink</td>
<td>Always 1.</td>
</tr>
<tr>
<td>st_size</td>
<td>Size of the file in bytes.</td>
</tr>
<tr>
<td>st_atime</td>
<td>Time of last modification of file (same as st_mtime and st_cftime).</td>
</tr>
<tr>
<td>st_mtime</td>
<td>Time of last modification of file (same as st_atime and st_cftime).</td>
</tr>
<tr>
<td>st_cftime</td>
<td>Time of last modification of file (same as st_atime and st_mtime).</td>
</tr>
</tbody>
</table>

There are three additional fields in the stat structure type that do not contain meaningful values under DOS.
### Return Value

The `fstat` function returns the value 0 if the file-status information is obtained. A return value of -1 indicates an error; in this case, `errno` is set to `EBADF`, indicating an invalid file handle.

### See Also

`access`, `chmod`, `filelength`, `stat`

---

**Note**

If the given `handle` refers to a device, the size and time fields in the `stat` structure are not meaningful.

---

### Example

```c
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <io.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";

main()
{
    fh = open("data", O_CREAT | O_WRONLY | O_TRUNC);
    write(fh, buffer, strlen(buffer));

    /* Get data associated with "fh": */
    result = fstat(fh, &buf);

    /* Check if statistics are valid: */
    if (result != 0)
        printf("Bad file handle\n");

    /* Output some of */
    /* the statistics: */
```
This program uses `fstat` to report the size of a file named `DATA`. 
Summary

#include <stdio.h>

long ftell(stream);
FILE *stream; Pointer to file structure

Description

The ftell function gets the current position of the file pointer (if any) associated with stream. The position is expressed as an offset relative to the beginning of the stream.

Return Value

The ftell function returns the current position. On error, the function returns -1L and errno is set to a nonzero value. On devices incapable of seeking (such as terminals and printers), or when stream does not refer to an open file, the return value is undefined.

See Also

fseek, lseek, tell

Note

The value returned by ftell may not reflect the physical byte offset for streams opened in text mode, since text mode causes carriage-return-line-feed translation. Use ftell in conjunction with the fseek function to remember and return to file locations correctly.
ftell

Example

#include <stdio.h>

FILE *stream;
long position;
char list[100];

main()
{
    stream = fopen("data","rb");

    /* Move the pointer by reading data: */
    fread(list,sizeof(char),100,stream);

    /* Get position after read: */
    position = ftell(stream);
    printf("position = %ld\n", position);
}

This program opens a file named data for reading and tries to read 100 characters. It then uses ftell to determine the position of the file pointer and displays this position.
### Summary

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

void ftime(timeptr);
struct timeb *timeptr; // Pointer to structure defined in sys/timeb.h
```

### Description

The `ftime` function gets the current time and stores it in the structure pointed to by `timeptr`. The `timeb` structure is defined in `sys/timeb.h`. It contains four fields, `time`, `millitm`, `timezone`, and `dstflag`, which have the following values:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.</td>
</tr>
<tr>
<td>millitm</td>
<td>Fraction of a second in milliseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>The difference in minutes, moving westward, between Greenwich mean time and local time. The value of <code>timezone</code> is set from the value of the global variable <code>timezone</code> (see <code>tzset</code>).</td>
</tr>
<tr>
<td>dstflag</td>
<td>Nonzero if daylight saving time is currently in effect for the local time zone, as determined from the value of the global variable <code>daylight</code> (see <code>tzset</code>).</td>
</tr>
</tbody>
</table>

### Return Value

The `ftime` function gives values to the fields in the structure pointed to by `timeptr`. It does not return a value.

### See Also

`asctime`, `ctime`, `gmtime`, `localtime`, `time`, `tzset`
ftime

Example

```c
#include <sys/types.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <time.h>

main()
{
    struct timeb timebuffer;
    char *timeline;

    ftime(&timebuffer);
    timeline = ctime(&timebuffer.time);

    printf("The time is %.19s.%hu %s", timeline,
            timebuffer.millitm, &timeline[20]);
}
```

Sample output:

```
The time is Wed Dec 04 17:58:29.420 1985
```

This program uses ftime to obtain the current time and then stores this time in timebuffer.
fwrite

■ Summary

#include <stdio.h>

int fwrite(buffer, size, count, stream);
char *buffer; Pointer to data to be written
int size; Item size in bytes
int count; Maximum number of items to be written
FILE *stream; Pointer to file structure

■ Description

The fwrite function writes as many as count items of length size from
buffer to the output stream. The file pointer associated with stream (if
there is one) is incremented by the number of bytes actually written.

If the given stream was opened in text mode, each carriage return is
replaced with a carriage-return–line-feed pair. The replacement has no
effect on the return value.

■ Return Value

The fwrite function returns the number of full items actually written,
which may be less than count if an error occurs. Also, if an error occurs,
the file position indicator is indeterminate.

■ See Also

fread, write
fwrite

- Example

```c
#include <stdio.h>

FILE *stream;
long list[100];
int numread;
int numwritten;

main()
{

    /* File opened in "binary" mode: */
    if (((stream = fopen("data.bin", "w+b")) != NULL)
    {
        /* Write 100 long integers to "stream": */
        numwritten = fwrite((char *)list,sizeof(long),100,stream);
        printf("Wrote %d items\n", numwritten);
    }
    else
        printf("Problem opening the file");

    fclose(stream);

    if (((stream = fopen("data.bin", "r+b")) != NULL)
    {
        /* Attempt to read in 100 long ints: */
        numread = fread((char *)list,sizeof(long),100,stream);
        printf("Number of items read = %d\n", numread);
    }
    else
        printf("Was not able to open the file");
}
```

This program opens a file named data.bin and writes 100 long integers to the file. The program then tries to open data.bin and read in 100 long integers. If the attempt succeeds, the program displays the number of actual items read.


\subsection*{gcvt}

\textbf{Summary}

\begin{verbatim}
#include <stdlib.h> Required only for function declarations

char *gcvt(value, ndec, buffer);
double value;
int ndec;
char *buffer;
\end{verbatim}

\textbf{Description}

The \texttt{gcvt} function converts a floating-point \texttt{value} to a character string and stores the string in \texttt{buffer}. The \texttt{buffer} should be large enough to accommodate the converted value plus a terminating null character ('\textbackslash 0'), which is automatically appended. There is no provision for overflow.

The \texttt{gcvt} function attempts to produce \texttt{ndec} significant digits in FORTRAN F format. Failing that, it produces \texttt{ndec} significant digits in FORTRAN E format. Trailing zeros may be suppressed in the conversion.

\textbf{Return Value}

The \texttt{gcvt} function returns a pointer to the string of digits. There is no error return.

\textbf{See Also}

\texttt{atof, atoi, atol, ecvt, fcvt}

\textbf{Example}

\begin{verbatim}
#include <stdlib.h>
#include <stdio.h>

char buffer[50];
int precision = 7;

main()
{
    gcvt(-3.1415e5, precision, buffer);
    /* buffer now contains "-314150." */
\end{verbatim}
printf( "buffer= \"%s\"\n", buffer );
}

This program converts \-3.1415e5 to its string representation, then displays this string.
getc – getchar

- Summary

#include <stdio.h>

int getc(stream);  // Read a character from stream
FILE *stream;      // Pointer to file structure

int getchar();     // Read a character from stdin

- Description

The getc macro reads a single character from the current stream position and increases the associated file pointer (if there is one) to point to the next character. The getchar macro is identical to getc(stdin).

- Return Value

The getc and getchar macros return the character read. A return value of EOF indicates an error or end-of-file condition. Use ferror or feof to determine whether an error or end-of-file occurred.

- See Also

fgetc, fgetchar, getch, getche, putc, putchar, ungetc

---

Note

The getc and getchar routines are identical to fgetc and fgetchar, but are macros, not functions.
Example

```c
#include <stdio.h>

FILE *stream;
char buffer[81];
int i, ch;

main()
{
    stream = fopen("getc.c", "r");

    printf("Enter a line >> ");

    /* Read in single line from "stdin": */
    for (i = 0; (i < 80) && ((ch = getchar()) != EOF)
        && (ch != 'n'); i++)
        buffer[i] = ch;

    /* Terminate string with null character: */
    buffer[i] = '\0';
    printf("%s\n", buffer);
}
```

This program uses `getchar` to read a single line of input from `stdin`, places this input in `buffer`, then terminates the string before printing it to the screen.
getch

- Summary

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

int getch();
```

- Description

The `getch` function reads, without echoing, a single character directly from the console. Characters typed are not echoed. The `getch` function cannot be used to read CTRL-C.

- Return Value

The `getch` function returns the character read. There is no error return.

- See Also

cgets, getche, getchar

- Example

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

int ch;

main()
{
    printf("Input whitespace characters, \nfollowed by a nonwhite space\n");
    do
    {
        ch = getch();
    } while (isspace(ch));
    putchar(ch);
}
```
This program reads characters from the keyboard, but does not echo them until it reads the first nonblank character.
getche

- **Summary**

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

int getche();
```

- **Description**

The `getche` function reads a single character from the console and echoes the character read. The `getche` function cannot be used to read a CTRL-C.

- **Return Value**

The `getche` function returns the character read. There is no error return.

- **See Also**

cgets, getch, getchar

- **Example**

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

int ch;

main()
{
    printf( "Type in upper- and lowercase letters\n" );
    do
    {
        ch = getche();
        if (isupper(ch))
            cprintf("\b%c", _tolower(ch));    /* Backspace and print */
        else
            cprintf("\b%c", _tolower(ch));    /* in lowercase */
    } while( ch != '\r' );
}
```

This program reads characters entered from the keyboard and echoes them on the screen. If a character is uppercase, the program converts it to lowercase and writes it over the old character. A carriage return terminates input.
getcwd

- **Summary**

```c
#include <direc.h>    // Required only for function declarations

char *getcwd(pathbuf, n);
char *pathbuf;        // Storage location for path name
int n;               // Maximum length of path name
```

- **Description**

The `getcwd` function gets the full path name of the current working directory and stores it at `pathbuf`. The integer argument `n` specifies the maximum length for the path name. An error occurs if the length of the path name (including the terminating null character) exceeds `n`.

The `pathbuf` argument can be `NULL`; a buffer of at least size `n` (more only if necessary) will automatically be allocated via `malloc` to store the path name. This buffer can later be freed by calling `free` and passing it the `getcwd` return value (a pointer to the allocated buffer).

- **Return Value**

The `getcwd` function returns `pathbuf`. A `NULL` return value indicates an error, and `errno` is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory to allocate <code>n</code> bytes (when <code>NULL</code> argument given as <code>pathbuf</code>)</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Path name longer than <code>n</code> characters</td>
</tr>
</tbody>
</table>

- **See Also**

`chdir`, `mkdir`, `rmdir`
getcwd

- Example

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

main()
{
    char buffer[67];

    /* Get the current working directory: */
    if (getcwd(buffer,66) == NULL)
        perror("getcwd error");
    else
        printf("%s",buffer);
}
```

This program places the name of the current directory in the buffer array, then displays the name of the current directory on the screen. Specifying a length of 66 characters for `buffer` allows room for the longest legal directory name plus two characters for the drive specification.
Summary

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

```c
char *getenv(varname);
const char *varname; Name of environment variable
```

Description

The `getenv` function searches the list of environment variables for an entry corresponding to `varname`. Environment variables define the environment in which a process executes (for example, the "LIB" environment variable defines the default search path for libraries to be linked with a program).

Return Value

The `getenv` function returns a pointer to the environment table entry containing the current string value of `varname`. The return value is `NULL` if the given variable is not currently defined.

See Also

`putenv`

Note

Environment table entries must not be changed directly. If an entry must be changed, use the `putenv` function. To modify the returned value without affecting the environment table, use `strdup` or `strcpy` to make a copy of the string.

The `getenv` and `putenv` functions use the global variable `environ` to access the environment table. The `putenv` function may change the value of `environ`, thus invalidating the `envp` argument to the `main` function. Therefore, it's safer to use the `environ` variable to access the environment information.
**getenv**

- **Example**

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

char *pathvar;

main()
{
    /* Get the value of the PATH */
    /* environment variable: */
    pathvar = getenv("PATH");

    printf("%s\n", pathvar ? pathvar : "path variable not set");
}
```

This program uses `getenv` to retrieve the `PATH` environment variable and then displays its value. If `PATH=A:\BIN;B:\BIN` is in the environment, `pathvar` points to `A:\BIN;B:\BIN`. If there is no `PATH` environment variable, `pathvar` is `NULL`. 
Summary

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

int getpid();

Description

The getpid function returns an integer, the process ID, that uniquely identifies the calling process.

Return Value

The getpid function returns the process ID. There is no error return.

See Also

mktemp

Example

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

char filename[9], pid[5];

main()
{
    strcpy(filename, "FILE");
    strcat(filename, itoa(getpid(), pid, 10));

    /* Prints "FILExxxx", where xxxx */
    /* is the process ID: */
    printf("Filename is %s\n", filename);
}

This program uses getpid to obtain the process ID, then converts the process ID to a string for output.
**gets**

- **Summary**

```c
#include <stdio.h>

char *gets(buffer);
char *buffer;
```

- **Description**

The `gets` function reads a line from the standard input stream `stdin` and stores it in `buffer`. The line consists of all characters up to and including the first new-line character ('\n'). The `gets` function then replaces the new-line character with a null character ('\0') before returning the line, unlike `fgets`, which retains the new-line character.

- **Return Value**

If successful, the `gets` function returns its argument. A NULL pointer indicates an error or end-of-file condition. Use `ferror` or `feof` to determine whether an error or end-of-file occurred.

- **See Also**

`fgets`, `fputs`, `puts`

- **Example**

```c
#include <stdio.h>

char line[100];
char *result;

main()
{
    printf("Input a string: ");
    result = gets(line);
    printf("The line entered was: %s\n", result);
}
```
This program uses `gets` to read a line of input from `stdin`. 
# Summary

```
#include <stdio.h>

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

**Description**

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 special alignment of items in the stream.

**Return Value**

The `getw` function returns the integer value read. A return value of `EOF` may indicate an error or end-of-file; however, the `EOF` value is also a legitimate integer value, so `feof` or `ferror` should be used to verify an end-of-file or error condition.

**See Also**

`putw`

---

**Note**

The `getw` function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with `getw` since the size of an `int` and ordering of bytes within an `int` differ across systems.
getw

Example

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

FILE *stream;
int i;

main()
{
stream = fopen("data.bin", "rb");

/* Read a word from the stream: */
i = getw(stream);

/* If there is an error... */
if (ferror(stream))
{
printf("getw failed\n");
clearerr(stream);
}
else
printf("Word = %x\n", i);
}

This program uses getw to read a word from a stream, then performs an error check.
# Summary

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

```c
struct tm *gmtime(time);
const time_t *time;  // Pointer to stored time
```

## Description

The `gmtime` function converts the `time` value to a structure. The `time` argument represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time; this value is usually obtained from a call to `time`.

The `gmtime` function breaks down the `time` value and stores it in a structure of type `tm`, defined in `time.h`. The structure result reflects Greenwich mean time, not local time.

The fields of the structure type `tm` store the following values:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tm_sec</code></td>
<td>Seconds</td>
</tr>
<tr>
<td><code>tm_min</code></td>
<td>Minutes</td>
</tr>
<tr>
<td><code>tm_hour</code></td>
<td>Hours (0–24)</td>
</tr>
<tr>
<td><code>tm_mday</code></td>
<td>Day of month (1–31)</td>
</tr>
<tr>
<td><code>tm_mon</code></td>
<td>Month (0–11; January = 0)</td>
</tr>
<tr>
<td><code>tm_year</code></td>
<td>Year (current year minus 1900)</td>
</tr>
<tr>
<td><code>tm_wday</code></td>
<td>Day of week (0–6; Sunday = 0)</td>
</tr>
<tr>
<td><code>tm_yday</code></td>
<td>Day of year (0–365; January 1 = 0)</td>
</tr>
<tr>
<td><code>tm_isdst</code></td>
<td>Nonzero if daylight saving time is in effect, otherwise 0</td>
</tr>
</tbody>
</table>

## Note

Under DOS, dates prior to 1980 are not understood. If `time` represents a date before January 1, 1980, `gmtime` returns the structure representation of 00:00:00, January 1, 1980.
gmtime

- Return Value

The **gmtime** function returns a pointer to the structure result. There is no error return.

- See Also

  asctime, ctime, ftim e, localtime, time

---

**Note**

The **gmtime** and **localtime** functions use a single statically allocated structure to hold the result. Each call to one of these routines destroys the result of the previous call.

---

- Example

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

struct tm *newtime;
long ltime;

main()
{
    time(&ltime);

    /* Obtain Greenwich mean time: */
    newtime = gmtime(&ltime);
    printf("Greenwich mean time is %s\n", asctime(newtime));
}
```

This program uses **gmtime** to convert a long-integer representation of Greenwich mean time to a structure named **newtime**, then uses **asctime** to convert this structure to an output string.
halloc

- Summary

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

cchar huge *halloc(n, size);
long n; // Number of elements
unsigned size; // Length in bytes of each element
```

- Description

The `halloc` function allocates storage space for a huge array of `n` elements, each of length `size` bytes. Each element is initialized to 0.

If the size of the array is greater than 128K (131,072), then the size of an array element must be a power of 2.

- Return Value

The `halloc` function returns a `char huge` pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than `char huge`, use a type cast on the return value. The return value is `NULL` if the request cannot be satisfied (for example, if not enough memory is available or if the huge array is specified illegally).

- See Also

calloc, free, hfree, malloc, realloc

- Example

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

main()
{
    long huge *lalloc;
    lalloc = (long huge *)halloc(30000L, sizeof(long));
    if (lalloc == NULL)
        printf("Insufficient memory available");
    else
        printf("Memory successfully allocated");
```
This program uses **halloc** to allocate space for 30,000 long integers, then initializes each element to 0.
hfree

- Summary

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

void hfree(ptr);
char huge *ptr;  // Pointer to allocated memory block
```

- Description

The `hfree` function deallocates a memory block; the free memory is returned to DOS. The `ptr` argument points to a memory block previously allocated through a call to `halloc`. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is available for allocation.

- Return Value

There is no return value.

- See Also

`halloc`

---

**Note**

Attempting to free an invalid `ptr` (a pointer not allocated with `halloc`) may affect subsequent allocation and cause errors.
Example

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

main()
{
    char huge *alloc;
    alloc = malloc(80000L, sizeof(char));

    /* Test for valid pointer: */
    if (alloc != NULL)
    {
        hfree(alloc);

        /* Free memory for the heap: */
        printf("Memory successfully allocated and deallocated");
    }
    else
        printf("Insufficient memory available");
}

This program allocates space for 80,000 characters and initializes this space to zeros, then uses hfree to deallocate that memory.
hypot

- Summary

```c
#include <math.h>

double hypot(x,y);

double x, y;               // Floating-point values
```

- Description

The `hypot` function calculates the length of the hypotenuse of a right triangle, given the length of the two sides `x` and `y`. A call to `hypot` is equivalent to the following:

```c
sqrt(x*x + y*y);
```

- Return Value

The `hypot` function returns the length of the hypotenuse. If an overflow results, `hypot` sets `errno` to `ERANGE` and returns the value `HUGE`.

- See Also

`cabs`

- Example

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

main()
{
    double x, y, z;
    x = 3.0;
    y = 4.0;

    z = hypot(x,y);        // z is 5.0 */
    printf("Hypotenuse = %2.1f\n", z);
}
```
Output:

Hypotenuse = 5.0

This program uses hypot to calculate the hypotenuse of a right triangle with sides of 3.0 and 4.0.
inp

- Summary

```c
#include <conio.h>  // Required only for function declarations
int inp(port);
unsigned port;    // Port number
```

- Description

The `inp` function reads 1 byte from the input port specified by `port`. The `port` argument can be any unsigned integer number in the range 0 to 65,535.

- Return Value

The `inp` function returns the byte read from `port`. There is no error return.

- See Also

`outp`

- Example

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

/* Read will be done on port #0: */
unsigned int port = 0;
char result;

main()
{
    /* Input a byte from the port: */
    result = inp(port);
    printf("The value from port #%d is %d\n", port, result);
}
```

This program reads a character from input port 0.


---

**Summary**

#include <dos.h>

int int86(intno, inregs, outregs);

- `int intno;` Interrupt number
- `union REGS *inregs;` Register values on call
- `union REGS *outregs;` Register values on return

**Description**

The int86 function executes the 8086 processor family interrupt specified by the interrupt number intno. Before executing the interrupt, int86 copies the contents of inregs to the corresponding registers. After the interrupt returns, the function copies the current register values to outregs. It also copies the status of the system carry flag to the cflag field in outregs. The inregs and outregs arguments are unions of type REGS. The union type is defined in the include file dos.h.

The int86 function is intended to be used to invoke DOS interrupts directly.

**Return Value**

The return value is the value in the AX register after the interrupt returns. If the cflag field in outregs is nonzero, an error has occurred and the _doserrno variable is also set to the corresponding error code.

**See Also**

bdos, intdos, intdosx, int86x
Example

#define VIDEO_IO 0x10
#define SET_CRSR 1

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

union REGS regs;

main()
{
    int top, bot;

    /* Get new cursor size from user: */
    printf("Enter new cursor top and bottom: ");
    scanf("%d %d", &top, &bot);

    /* Set up for cursor change call: */
    regs.h.ah = SET_CRSR;
    regs.h.ch = top;
    regs.h.cl = bot;

    /* Execute interrupt: */
    int86(VIDEO_IO, &regs, &regs);
}

This program uses int86 to call the IBM-PC BIOS video service (INT 10H) to change the size of the cursor.

The default values are:

    Monochrome card: 12 13
    Color Card : 6 7
    43-line EGA : 4 5
Summary

#include <dos.h>

int int86x(intno, inregs, outregs, segregs);

int intno;
union REGS *inregs; Interrupt number
union REGS *outregs; Register values on call
union REGS *segregs; Register values on return
struct SREGS *segregs; Segment-register values on call

Description

The int86x function executes the 8086 processor family interrupt specified by the interrupt number intno. Unlike the int86 function, int86x accepts segment-register values in segregs, letting programs that use large-model data segments or far pointers specify which segment or pointer should be used during the system call.

Before executing the specified interrupt, int86x copies the contents of inregs and segregs to the corresponding registers. Only the DS and ES register values in segregs are used. After the interrupt returns, the function copies the current register values to outregs, copies the current ES and DS values to segregs, and restores DS. It also copies the status of the system carry flag to the cflag field in outregs. The inregs and outregs arguments are unions of type REGS. The segregs argument is a structure of type SREGS. These types are defined in the include file dos.h.

The int86x function is intended to be used to directly invoke DOS interrupts that take an argument in the ES register, or take a DS register value that is different from the default data segment.

Return Value

The return value is the value in the AX register after the interrupt returns. If the flag field in outregs is nonzero, an error has occurred and the doserrno variable is also set to the corresponding error code.
int86x

- See Also

bdos, intdos, intdosx, int86, segread, FP_SEG

---

Note

Segment values for the `segregs` argument can be obtained by using either the `segread` function or the `FP_SEG` macro.

---

- Example

```c
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

#define SYSCALL 0x21  /* INT 21H invokes system calls */
#define CHANGE_ATTR 0x43  /* system call 43H */
    /* actually changes attributes */

char far *filename = "int86x.c";  /* filename in 'far' */
    /* data segment */

union REGS inregs, outregs;
struct SREGS segregs;
int result;

main()
{

** AH us system call number
** AL is function (get attributes)
** DS:DX points to file name
*/

inregs.h.ah = CHANGE_ATTR;
inregs.h.al = 0;
inregs.x.dx = FP_OFF(filename);
segregs.ds = FP_SEG(filename);
result = int86x(SYSCALL, &inregs, &outregs, &segregs);
if (outregs.x.cflag)
{
    printf("Can't get file attributes; error no. %d\n", result);
    exit(1);
}
```
else
    printf("Attrs = %#x\n", outregs.x.cx);
}

In this program, **int86x** executes an INT 21H instruction to invoke DOS system call 43H (change file attributes). In this case, **int86x** is used because the file, which is referenced with a *far* pointer, may be in a segment other than the default data segment. Thus, the program must explicitly set the *DS* register with the *SREG* structure.
intdos

• Summary

#include <dos.h>

int intdos(inregs, outregs);
union REGS *inregs; Register values on call
union REGS *outregs; Register values on return

• Description

The intdos function invokes the DOS system call specified by register values defined in inregs and returns the effect of the system call in outregs. The inregs and outregs arguments are unions of type REGS. The union type is defined in the include file dos.h.

To invoke a system call, intdos executes an INT 21H instruction. Before executing the instruction, the function copies the contents of inregs to the corresponding registers. After the INT instruction returns, intdos copies the current register values to outregs. It also copies the status of the system carry flag to the cflag field in outregs. If this field is nonzero, the flag was set by the system call and indicates an error condition.

The intdos function is intended to be used to invoke DOS system calls that take arguments in registers other than DX (DH/DL) and AL, or to invoke system calls that indicate errors by setting the carry flag.

• Return Value

The intdos function returns the value of the AX register after the system call is completed. If the cflag field in outregs is nonzero, an error has occurred and _doserrno is also set to the corresponding error code.

• See Also

bdos, intdosx
Example

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

union REGS inregs, outregs;

main()
{
  /* Setup for function call 2a hex: */
  inregs.h.ah = 0x2a;

  /* Get current date: */
  intdos(&inregs,&outregs);
  printf("date is %d/%d/%d\n",
         outregs.h.dh,outregs.h.dl,outregs.x.cx);
}
```

This program uses `intdos` to invoke DOS system call 2AH (get the current date).
intdosc

• Summary

#include <dos.h>

int intdosc(intregs, outregs, segregs);
union REGS *inregs; Register values on call
union REGS *outregs; Register values on return
struct SREGS *segregs; Segment-register values on call

• Description

The intdosc function invokes 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, intdosc accepts segment-register values in segregs, letting programs that use long-model data segments or far pointers specify which segment or pointer should be used during the system call. The inregs and outregs arguments are unions of type REGS. The segregs argument is a structure of type SREGS. These types are defined in the include file dos.h.

To invoke a system call, intdosc executes an INT 21H instruction. Before executing the instruction, the function copies the contents of inregs and segregs to the corresponding registers. Only the DS and ES register values in segregs are used. After the INT instruction returns, intdosc copies the current register values to outregs and restores DS. It also copies the status of the system carry flag to the cflag field in outregs. If this field is nonzero, the flag was set by the system call and indicates an error condition.

The intdosc function is intended to be used to invoke DOS system calls that take an argument in the ES register, or that take a DS register value different from the default data segment.

• Return Value

The intdosc function returns the value of the AX register after the system call is completed. If the cflag field in outregs is nonzero, an error has occurred and _doserrno is also set to the corresponding error code.
See Also

bdos, intdos, segread, FP_SEG

Note

Segment values for the segregs argument can be obtained by using either the segread function or the FP_SEG macro.

Example

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

union REGS inregs, outregs;
struct SREGS segregs;
char buffer[51], buf2[51]; /* Buffers for directory names */
char far *dir = "\newdir"; /* Directory to create */
char *result1, result2;

main()
{
    result1 = getcwd(buffer,50);
    printf("Current working directory is =\%s\n", buffer);

    mkdir(dir);

    inregs.h.ah = Ox3b; /* Change directory function */
inregs.x.dx = FP_OFF(dir); /* File name offset */
segregs.ds = FP_SEG(dir); /* File name segment */
intdosx(&inregs,&outregs,&segregs);

    result1 = getcwd(buf2,50);
    printf("Changed working directory is =\%s\n", buf2);
    result2 = chdir(buffer); /* Change back */
    result1 = getcwd(buf2, 50);
    printf("Changed to original working directory =\%s\n", buf2);
}
```

First, this program gets and displays the name of the current directory and creates a directory named \NEWDIR on the current drive. Then it invokes DOS system call 3BH using intdosx to change the current working directory to \NEWDIR. Finally, it restores the original directory as the
intdosx

current working directory.
isalnum – isascii

- Summary

```c
#include <ctype.h>

int isalnum(c);    // Test for alphanumeric ('A'-'Z', 'a'-'z', or '0'-'9')
int isalpha(c);    // Test for letter ('A'-'Z' or 'a'-'z')
int isascii(c);    // Test for ASCII character (0x00-0x7F)
int c;            // Integer to be tested
```

- Description

The `ctype` routines listed above test a given integer value, returning a nonzero value if the integer satisfies the test condition and a 0 value if it does not. An ASCII character set environment is assumed.

The `isascii` routine produces meaningful results for all integer values. However, the remaining routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where `isascii` holds true) or for the non-ASCII value EOF (defined in `stdio.h`).

- See Also

-iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, tolower, toupper

---

**Note**

The `ctype` routines are implemented as macros.
isalnum - isascii

Example

#include <stdio.h>
#include <ctype.h>

main()
{
    int ch;
    for (ch = 0; ch <= 0x7f; ch++)
    {
        printf("%#04x", ch);
        printf("%3s", isalnum(ch) ? "AN" : "");
        printf("%2s", isalpha(ch) ? "A" : "");
        printf("%3s", isascii(ch) ? "AS" : "");

        putchar(\n');
    }
}

This program uses isalnum, isalpha, and isascii to test all characters between 0x0 and 0x7f. It displays each character tested, followed by a code indicating the character type: A for alpha characters, AN for alphanumeric characters, or AS for ASCII characters.
isatty

■ Summary

#include <io.h>  

int isatty(handle);
int handle;

■ Description

The isatty function determines whether the given handle is associated with a character device (that is, a terminal, console, printer, or serial port).

■ Return Value

The isatty function returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

■ Example

#include <io.h>

int fh;
long loc;

main()
{
   int interactive;
   fh = fopen("isatty.c", "r");
   interactive = isatty(fh);
   printf("Is file handle a device? %s.\n", interactive ? "yes" : "no");
   /* if not a character device, get current position: */
   if (isatty(fh) == 0)
      loc = tell(fh);
}

This program opens a file named isatty.c for input and uses the isatty function to determine whether isatty.c is a device. If not, the program displays the current position.
iscntrl – isxdigit

Summary

#include <ctype.h>

iscntrl(c);  // Test for control character (0x00–0x1f or 0x7f)
isdigit(c);  // Test for digit ('0'–'9')
isgraph(c);  // Test for printable character not including the space character (0x21–0x7e)
islower(c);  // Test for lowercase ('a'–'z')
isprint(c);  // Test for printable character (0x20–0x7e)
ispunct(c);  // Test for punctuation character (isalnum(c), iscntrl(c), and isspace(c) all false)
isspace(c);  // Test for white-space character (0x09–0x0d or 0x20)
isupper(c);  // Test for uppercase ('A'–'Z')
isxdigit(c); // Test for hexadecimal digit ('A'–'F', 'a'–'f', or '0'–'9')

int c;     // Integer value to be tested

Description

The ctype routines listed above test a given integer value, returning a nonzero value if the integer satisfies the test condition, and 0 if it does not. An ASCII character set environment is assumed.

These routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where isascii holds true) or for the non-ASCII value EOF (defined in stdio.h).
iscntrl — isxdigit

- See Also

isalnum, isalpha, isascii, toascii, tolower, toupper

---

Note

The `ctype` routines are implemented as macros.

---

- Example

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

main ()
{
  int ch;
  for (ch = 0; ch <= 0x7f; ch++)
  {
    printf("%2s", iscntrl(ch) ? "C" : "");
    printf("%2s", isdigit(ch) ? "D" : "");
    printf("%2s", isgraph(ch) ? "G" : "");
    printf("%2s", islower(ch) ? "L" : "");
    printf("%c", isprint(ch) ? ch : '\0');
    printf("%3s", ispunct(ch) ? "PU" : "");
    printf("%2s", isspace(ch) ? "S" : "");
    printf("%3s", isprint(ch) ? "PR" : "");
    printf("%2s", isupper(ch) ? "U" : "");
    printf("%2s", isxdigit(ch) ? "X" : "");

    putchar('\n');
  }
}
```

This program tests all characters between 0x0 and 0x7f, then displays each character with any of the following character-type codes that apply:

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Control</td>
</tr>
<tr>
<td>D</td>
<td>Digit</td>
</tr>
<tr>
<td>G</td>
<td>Graphics</td>
</tr>
<tr>
<td>----</td>
<td>-----------</td>
</tr>
<tr>
<td>L</td>
<td>Lowercase</td>
</tr>
<tr>
<td>PR</td>
<td>Printable</td>
</tr>
<tr>
<td>S</td>
<td>Space</td>
</tr>
<tr>
<td>PU</td>
<td>Punctuation</td>
</tr>
<tr>
<td>U</td>
<td>Uppercase</td>
</tr>
<tr>
<td>X</td>
<td>Hexadecimal digit</td>
</tr>
</tbody>
</table>

The program prints all printable characters in the tested range.
itoa

- **Summary**

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

char *itoa(value, string, radix);
```

- **Description**

The `itoa` function converts the digits of the given `value` to a null-terminated 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 (−).

- **Return Value**

The `itoa` function returns a pointer to `string`. There is no error return.

- **See Also**

`itoa`, `ultoa`

---

**Note**

The space allocated for `string` must be large enough to hold the returned string. The function can return up to 17 bytes.

---

- **Example**

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

int radix = 8;
char buffer[20];
char *p;
```
main()
{
    p = itoa(-3445, buffer, radix);    /* p = "171213" */
    printf( "buffer= \"%s\"\n", buffer );
}

This program converts the integer -3445 to a string using a radix of 8 and displays that string.
# Summary

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

int kbhit();
```

## Description

The `kbhit` function checks the console for a recent keystroke.

## Return Value

The `kbhit` function returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

## Example

```c
main()
{
    printf( "waiting...
" );

    // Loop until kbhit() reports a keystroke:
    while( !kbhit() ){
        printf( "key struck was '\%c', getc() ");
    }
}
```

This program loops until the user presses a key. If result from `kbhit` is nonzero, a keystroke is waiting in the buffer. The program can use the `getc` or `getche` function to fetch the keystroke. If the program calls `getc` or `getche` without first checking `kbhit`, the program might pause while waiting for input.
labs

- **Summary**

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

long labs(n);
long n;              // Long integer value
```

- **Description**

The `labs` function produces the absolute value of its long-integer argument `n`.

- **Return Value**

The `labs` function returns the absolute value of its argument. There is no error return.

- **See Also**

`abs, cabs, fabs`

- **Example**

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

main()
{
    long x, y;
    x = -41567L;
    y = labs(x);
    printf("The labs(%ld) = %ld", x, y);
}
```

This program uses `labs` to get the absolute value of -41567, then displays this absolute value.
# Summary

```c
#include <math.h>

double ldexp(x, exp);
```

- `x`: Floating-point value
- `exp`: Integer exponent

## Description

The `ldexp` function calculates the value of `x * 2^exp`.

## Return Value

The `ldexp` function returns `x * 2^exp`. If an overflow results, the function returns ±HUGE (depending on the sign of `x`) and sets `errno` to `ERANGE`.

## See Also

`frexp`, `modf`

## Example

```c
#include <math.h>

main()
{
    double x, y;
    int p;
    x = 1.5;
    p = 5;
    y = ldexp(x, p);
    /* y = 48.0 */
    printf("The ldexp(%f,%d) = %f", x, p, y);
}
```

This program uses `ldexp` to calculate the value of `1.5 * 2^5`.
ldiv

- Summary

#include <stdlib.h>

ldiv_t ldiv(numer, denom);
long int numer; Numerator
long int denom; Denominator

- Description

The ldiv function divides numer by denom, computing the quotient and
the remainder. Where the division cannot be performed inexactely, the quo-
tient is returned with the proper sign and it's value is the largest integer
value less than the exact mathematical value. The behavior of ldiv is
undefined if the result cannot be represented.

The ldiv function is similar to the div function, the difference being that
the arguments and the members of the returned structure are all of type
long int.

- Return Value

The ldiv function returns a structure of type ldiv_t, comprising both the
quotient and the remainder. The structure contains the following
members:

long int quot; Quotient
long int rem; Remainder

- See Also

div

- Example

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

main(argc, argv)
    int argc;
char **argv;
{
long int x,y;
ldiv_t div_result;
x = atol(argv[1]);
y = atol(argv[2]);
printf("x is \%d, y is \%d, x,y\);
div_result = ldiv(x,y);
printf("The quotient is \%d, and the remainder is \%d,\n
div_result.quot, div_result.rem);\n}\

This program accepts two arguments on the command line, then calls ldiv to divide the first argument by the second. Finally, it prints the structure members quot and rem.
lfind – lsearch

- Summary

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

char *lsearch(key, base, num, width, compare);

char *lfind(key, base, num, width, compare);

description

The lsearch and lfind functions perform a linear search for the value key in an array of num elements, each of width bytes in size. (Unlike bsearch, lsearch and lfind do not require the array to be sorted.) The argument base is a pointer to the base of the array to be searched.

If the key is not found, lsearch adds it to the end. The lfind function does not.

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not equal to 0</td>
<td>element1 and element2 different</td>
</tr>
<tr>
<td>0</td>
<td>element1 identical to element2</td>
</tr>
</tbody>
</table>

- Return Value

If the key is found, both lsearch and lfind return a pointer to the array element that base points to. If the key is not found, lfind returns NULL, and lsearch returns a pointer to a newly added item at the end of the array.
See Also
bsearch

Example

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

int compare(); /* must declare as a function */

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

    result = (char **)lfind((char *)&key, (char *)argv, &argc,
                           sizeof(char *), compare);
    if (result)
        printf("%s found\n", *result);
    else
        printf("PATH not found!\n");
}

int compare (arg1, arg2)
char **arg1, **arg2;
{
    return (strncmp(*arg1,*arg2,strlen(*arg1)));}
```

This program uses `lfind` to search for the key word `PATH` in the command-line arguments. Unlike `lsearch`, `lfind` fails if the key word is not found.
localtime

- Summary

```
#include <time.h>

struct tm *localtime(time);
long *time;                  // Pointer to stored time
```

- Description

The `localtime` function converts a time stored as a `long` value to a structure. The `long` value `time` represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time; this value is usually obtained from the `time` function.

The `localtime` function breaks down the `time` value, corrects for the local time zone and daylight saving time if appropriate, and stores the corrected time in a structure of type `tm`. (See `gmtime` for a description of the `tm` structure fields.)

Under DOS, dates prior to 1980 are not understood. If `time` represents a date before January 1, 1980, `localtime` returns the structure representation of 00:00:00 January 1, 1980.

The `localtime` function makes corrections for the local time zone if the user first sets the environment variable `TZ`. The value of `TZ` must be a three-letter time zone name (such as PST), followed by a possibly signed number giving the difference between Greenwich mean time and the local time zone. The number may be followed by a three-letter daylight saving time zone (such as PDT). The `localtime` function uses the difference between Greenwich mean time and local time to adjust the stored time value. If a daylight saving time zone is present in the `TZ` setting, `localtime` also corrects for daylight saving time. If `TZ` currently has no value, the default value PST8PDT is used.

When `TZ` is set, three other environment variables, `timezone`, `daylight`, and `tzname`, are automatically set as well. See the `tzset` function for a description of these variables.
Return Value

The `localtime` function returns a pointer to the structure result. There is no error return.

See Also

`asctime`, `ctime`, `ftime`, `gmtime`, `time`, `tzset`

Note

The `gmtime` and `localtime` functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

Example

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

struct tm *newtime;
long ltime;

main()
{
    struct tm *newtime;
    char *am_pm = "PM";
    time_t long_time;

    time(&long_time); /* Get time as long integer */
    newtime = localtime(&long_time); /* Convert to local time */

    if (newtime->tm_hour < 12) /* Set up extension */
        am_pm = "AM";
    if (newtime->tm_hour > 12) /* Convert from 24 hour */
        newtime->tm_hour -= 12; /* to 12 hour clock */

    printf("%.19s %s\n", asctime(newtime), am_pm);
}
```
localtime

Sample output:

Tue Dec 10 11:30:12 AM

This program uses `time` to get the current time and then uses `localtime` to convert this time to a structure representing the local time. The program converts the result from a 24-hour clock to a 12-hour clock and determines the proper extension (AMorPM).
locking

Summary

```
#include <sys\locking.h>
#include <io.h>

int locking(handle, mode, nbyte);
int handle; File handle
int mode; File locking mode
long nbyte; Number of bytes to lock
```

Description

The **locking** function locks or unlocks *nbyte* bytes of the file specified by *handle*. Locking bytes in a file prevents subsequent reading and writing of those bytes by other processes. Unlocking a file permits other processes to read or write to previously locked bytes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbyte* bytes, or to the end of the file.

---

**Important**

Under DOS Versions 3.0 and 3.1, a parent process's locked files may become unlocked when one of its children exits.

---

The argument *mode* specifies the locking action to be performed. It must be one of the following manifest constants:

<table>
<thead>
<tr>
<th>Manifest Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LK_LOCK</td>
<td>Lock the specified bytes. If the bytes cannot be locked, try again after 1 second. If, after 10 attempts, the bytes cannot be locked, return an error.</td>
</tr>
<tr>
<td>LK_RLCK</td>
<td>Same as LK_LOCK.</td>
</tr>
<tr>
<td>LK_NBLCK</td>
<td>Lock the specified bytes. If bytes cannot be locked, return an error.</td>
</tr>
</tbody>
</table>
locking

LK_NBRLCK  Same as LK_NBLCK.
LK_UNLCK  Unlock the specified bytes. The bytes must have been previously locked.

More than one region of a file can be locked, but no overlapping regions are allowed. Furthermore, no more than one region can be unlocked at a time.

When unlocking a file, the region of the file being unlocked must correspond to a region that was previously locked. The locking function does not coalesce adjacent regions, so if two locked regions are adjacent, each region must be unlocked separately.

All locks should be removed before closing a file or exiting the program.

■ Return Value

The locking function returns 0 if it is successful. A return value of -1 indicates failure, and errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCESS</td>
<td>Locking violation (file already locked or unlocked).</td>
</tr>
<tr>
<td>EBADF</td>
<td>Invalid file handle.</td>
</tr>
<tr>
<td>EDEADLOCK</td>
<td>Locking violation. This is returned when the LK_LOCK or LK_RLCK flag is specified and the file cannot be locked after 10 attempts.</td>
</tr>
</tbody>
</table>

EINVAL

■ See Also

creat, open

---

Note

The locking function should be used only under DOS 3.0 and later; it has no effect under earlier versions of DOS.
locking

- Example

```c
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/locking.h>
#include <stdlib.h>

extern unsigned char _osmajor;
int fh;
long pos;
char buffer[BUFSIZ];

main()
{
    int result;

    /* save the current file pointer position,
    ** then lock a region from the beginning of
    ** the file to the saved file pointer
    ** position
    */

    /* Open file; read 10 bytes: */
    fh = open("data", O_RDONLY);
    result = read(fh, buffer, 10);
    if (_osmajor >= 3) /* Check for DOS version >= 3.0 */
    {
        pos = tell(fh); /* Get current pointer position*/

        /* Reset pointer to beginning of file: */
        result = lseek(fh, 0L, SEEK_SET);

        /* Lock first portion of the file: */
        if ((locking(fh, LK_NBLCK, pos)) != -1)
        {
            printf("Successfully locked %d bytes\n", pos);
            lseek(fh, 0L, 0);
            locking(fh, LK_UNLCK, pos);
        }
        else
            perror("Locking failed");
    }
    else
        printf("DOS version must be 3, or higher.\n" );
}
```
This program opens a file named DATA and reads the first 10 bytes from the file. It then moves the file pointer back to the beginning of the file and uses locking to lock the first 10 bytes of the file.

Note that this program works correctly only if the following conditions are met:

- The file named DATA exists.
- SHARE.COM or SHARE.EXE is installed.
- The program is run under DOS Version 3.0 or later.
log – log10

• Summary

#include <math.h>

double log(x); Calculate natural logarithm of x
double log10(x); Calculate logarithm base 10 of x
double x; Floating-point value

• Description

The log and log10 functions calculate the natural logarithm and base 10 logarithm of x, respectively.

• Return Value

The log and log10 functions return the logarithm result. If x is negative, both functions print a DOMAIN error message to stderr and return the value negative HUGE. If x is 0, both functions print a SING error message and return the value negative HUGE. In either case, errno is set to EDOM.

Error handling can be modified by using the matherr routine.

• See Also

exp, matherr, pow

• Example

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

main()
{
    double x = 1000.0, y;

    y = log(x); /* y is 6.907755 */
    printf("The log(%.2f) = \%f\n", x, y);

    y = log10(x); /* y is 3.0 */
\begin{verbatim}

    printf("The log10(%.2f) = %f\n",x,y);
}

This program uses \texttt{log} and \texttt{log10} to calculate the natural logarithm and the base-10 logarithm of 1000, respectively.
\end{verbatim}
longjmp

- Summary

```c
#include <setjmp.h>

void longjmp(env, value);
jmp_buf env;                      // Variable in which environment is stored
int value;                       // Value to be returned to setjmp call
```

- Description

The `longjmp` function restores a stack environment previously saved in `env` by `setjmp`. The `setjmp` and `longjmp` functions provide a way to execute a nonlocal goto and are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to `setjmp` causes the current stack environment to be saved in `env`. A subsequent call to `longjmp` restores the saved environment and returns control to the point immediately following the corresponding `setjmp` call. Execution resumes as if the given `value` had just been returned by the `setjmp` call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when `longjmp` was called. The values of register variables are unpredictable.

The `longjmp` function must be called before the function that called `setjmp` returns. If `longjmp` is called after the function calling `setjmp` returns, unpredictable program behavior will result.

The `value` returned by `longjmp` must be nonzero. If a 0 argument is given for `value`, the value 1 is substituted in the actual return.

- Return Value

There is no return value.

- See Also

`setjmp`
Warning

The values of register variables in the routine calling setjmp may not be restored to the proper values after a longjmp is executed.

Example

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

jmp_buf mark;

main()
{
    if (setjmp(mark) != 0)    /* Set the point to jump to */
    {
        printf("longjmp has been called\n");
        recover();
        exit(1);
    }
    printf("setjmp has been called\n");
    p();
}

p()
{
    /* Routine to trigger an error */
    int error = 0;
    error = 1;
    if (error != 0)
        longjmp(mark,-1);    /* Execute a long jump */
}

recover()    /* Code goes here for recovery */
{
    /* from the error */
    /* Ensure that data files won't be corrupted by
     * exiting the program */
}
```

This program uses setjmp to save the stack environment and executes the p function to simulate an error. It then uses longjmp to restore the stack environment and resume execution immediately after the setjmp call. Because longjmp and setjmp return different values, a conditional expression in the program allows the program to call the recover
longjmp

c function to use additional error-recovery code.
lseek

- Summary

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

long lseek(handle, offset, origin);
```

- Description

The **lseek** function moves the file pointer (if any) associated with `handle` to a new location that is `offset` bytes from the `origin`. The next operation on the file occurs at the new location. The `origin` must be one of the following constants defined in `stdio.h`:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>Beginning of file</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Current position of file pointer</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>End of file</td>
</tr>
</tbody>
</table>

The **lseek** function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However, an attempt to position the pointer before the beginning of the file causes an error.

- Return Value

The **lseek** function returns the `offset`, in bytes, of the new position from the beginning of the file. A return value of -1L indicates an error, and `errno` is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid value for <code>origin</code>, or position specified by <code>offset</code> is before the beginning of the file</td>
</tr>
</tbody>
</table>
On devices incapable of seeking (such as terminals and printers), the return value is undefined.

- See Also

fseek, tell

- Example

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

int fh;
long pos; /* position of file pointer */
char buffer[10];

main()
{
    fh = open("data", O_RDONLY);

    /* Seek the beginning of the file: */
    pos = lseek(fh, OL, SEEK_SET);
    if (pos == -1L)
        perror("lseek to beginning failed");
    else
        printf("Position for beginning of file seek = %ld\n", pos);
    read(fh, buffer, 10); /* Move file pointer a little */

    /* Find current position: */
    pos = lseek(fh, 0L, SEEK_CUR);
    if (pos == -1L)
        perror("lseek to current position failed");
    else
        printf("Position for current position seek = %ld\n", pos);

    /* Set the end of the file: */
    position = lseek(fh, OL, SEEK_END);
    if (position == -1L)
        perror("lseek to end failed");
    else
        printf("Position for end of file seek = %ld\n", pos);
}
```
lseek

This program first opens a file named DATA. It then uses lseek to find the beginning of the file, to find the current position in the file, and to find the end of the file.
### Summary

```c
#include <stdlib.h>
char *ltoa(value, string, radix);
```

- **value**: Number to be converted
- **string**: String result
- **radix**: Base of `value`

### Description

The `ltoa` function converts the digits of the given `value` to a null-terminated 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 (`-`).

### Return Value

The `ltoa` function returns a pointer to `string`. There is no error return.

### See Also

`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.

---

### Example

```c
#include <stdlib.h>

int radix = 10;
char buffer[20];
char *p;
```
ltoa

main()
{
    p = ltoa(-344115L, buffer, radix); /* p = "-344115" */
    printf("Buffer= \"%s\"\n", buffer);
}

This program converts the long integer -34415 to a string with a base of radix. Note that since radix is 10, the first character in the buffer is a minus sign (-).
malloc

- Summary

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

void *malloc(size);
size_t size;        Bytes in allocated block

- Description

The malloc function allocates a memory block of at least size bytes. (The block may be larger than size bytes, due to space required for alignment and for maintenance information.)

If size is 0, malloc allocates a zero-length item (that is, a header only) in the heap. The resulting pointer can be passed to the realloc function to adjust the size at any time.

In near data models (small and medium model), malloc maps to _nmalloc. In far data models (compact and large model), malloc maps to _fmalloc.

- Return Value

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

- See Also

calloc, free, realloc

- Example

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

int *intarray;

main()
{
    /* Allocate space for 20 integers: */
    intarray = (int *)malloc(20*sizeof(int));

    if (intarray == NULL )
        printf( "Insufficient memory available\n" );
    else
        printf( "Memory space allocated for 20 integers.\n" );
}

This program uses malloc to allocate space from the heap for 20 integers.
matherr

- Summary

#include <math.h>

int matherr(x);
struct exception *x;  // Math exception information

- Description

The matherr function processes errors generated by the functions of the math library. The math functions call matherr whenever an error is detected. The user can provide a different definition of the matherr function to carry out special error handling.

When an error occurs in a math routine, matherr is called with a pointer to the following structure (defined in math.h) as an argument:

struct exception {
    int type;
    char *name;
    double arg1, arg2, retval;
};

The type specifies the type of math error. It will be one of the following values, defined in math.h:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMAIN</td>
<td>Argument domain error</td>
</tr>
<tr>
<td>SING</td>
<td>Argument singularity</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>Overflow range error</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>Underflow range error</td>
</tr>
<tr>
<td>TLOSS</td>
<td>Total loss of significance</td>
</tr>
<tr>
<td>PLOSS</td>
<td>Partial loss of significance</td>
</tr>
</tbody>
</table>

The structure member name is a pointer to a null-terminated string containing the name of the function that caused the error. The structure members arg1 and arg2 specify the values that caused the error. (If only one argument is given, it is stored in arg1.)
The default return value for the given error is **retval**. You can change this return value; keep in mind that the return value must specify whether or not an error actually occurred. If **matherr** returns 0, an error message is displayed and **errno** is set to an appropriate error value. If **matherr** returns a nonzero value, no error message is displayed and **errno** remains unchanged.

### Return Value

The **matherr** function should return 0 to indicate an error, and nonzero to indicate successful corrective action.

### See Also

**acos, asin, atan, atan2, bessel, cabs, cos, cosh, exp, hypot, log, pow, sin, sinh, sqrt, tan**

### Example

```c
#include <math.h>
#include <string.h>

int matherr(x)
struct exception *x;
{
    /* If the error is a domain error... */
    if (x->type == DOMAIN )
    {
        /* ... and the function name is "log" */
        if (strcmp(x->name, "log") == 0)
        {
            x->retval = log(-x->argl);
            return(1);
        }
    else

        /* If the function is "log10" */
        if ( strcmp(x->name, "log10") == 0 )
        {
            x->retval = log10(-x->argl);
            return(1);
        }
    }
```
matherr

    return(0);   /* Else the default actions are used */
}

main()
{
    printf("log(-2)=%e, log10(-5)=%e, log(0)=%e
",  
           log(-2.0), log10(-5), log(0));
}

This program calls a series of math functions. If any of these functions causes an error, the program calls matherr. If the error resulted from a negative argument to log or log10 (a domain error), the program returns the natural or base-10 logarithm of the absolute value of the argument and suppresses the usual error message. Otherwise, the program takes the usual default actions.
# Summary

**# include <malloc.h>**

Required only for function declarations

unsigned int _memavl();

**Description**

The **_memavl** function returns the approximate size, in bytes, of the memory available for dynamic memory allocation in the default data segment. This function can be used with **_calloc, malloc, or realloc** in the small and medium memory models, and with **_nmalloc** in all memory models.

**Return Value**

The **_memavl** function returns the size in bytes as an unsigned integer.

**See Also**

**_calloc, malloc, _freect, realloc, stackavail**

**Example**

```c
#include <malloc.h>

main()
{
    long *longptr;

    printf("Memory available before malloc = %u\n", _memavl());
    longptr = (long*)malloc(5000*sizeof(long));
    printf("Memory available after malloc = %u\n", _memavl());
}
```

Sample output:

```
Memory available before malloc = 61383
Memory available after malloc = 40959
```
_memavl_

This program uses _memavl_ to determine the amount of available memory. It then uses malloc to allocate space for 5000 long integers and uses _memavl_ again to determine the new amount of available memory.
memccpy

Summary

#include <memory.h> Required only for function declarations
#include <string.h> Use either string.h or memory.h

char *memccpy(dest, src, c, cnt);
char *dest; Pointer to destination
char *src; Pointer to source
int c; Last character to copy
unsigned cnt; Number of characters

Description

The memccpy function copies 0 or more bytes of src to dest, copying up to and including the first occurrence of the character c or until cnt bytes have been copied, whichever comes first.

Return Value

If the character c is copied, memccpy returns a pointer to the byte in dest that immediately follows the character. If c is not copied, memccpy returns NULL.

See Also

memchr, memcmp, memcpy, memset

Example

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

char buffer[100], source[100] = "This is the \ string to be transferred\n";
char *result;

main()
{
    result = memccpy(buffer, source, '\n',100);
    if (result == NULL)
memccpy

    printf("Memory has been copied, but no \n was found");
else
    printf("Memory has been copied, and a \n was found");
}

This program uses memccpy to copy a string from source to buffer. The copy proceeds until either 100 bytes have been copied or a new-line character (\n) is encountered.
# Summary

\# include <memory.h>  
\# include <string.h>  

void *memchr(buf, c, cnt);  
void *buf;  
size_t c;  
unsigned cnt;

• Description

The memchr function searches the first count bytes of buf for the first occurrence of the character c. The search continues until c is found or cnt bytes have been examined.

• Return Value

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.

• See Also

memccpy, memcmp, memcpy, memset

• Example

\# include <memory.h>  
\# include <stdio.h>

char buffer[100];  
char *result;

main()
{
    strcpy(buffer, "this is a test");

    result = memchr(buffer, 'a', 100);  
    if (result != NULL)
        printf("Char. 'a' found at position %d\n", result-buffer+1);
    else
        printf("Char 'a' not found in first 100 bytes of buffer");
This program uses `memchr` to find the first occurrence of a in the buffer. If a is not in the first 100 bytes, `memchr` returns NULL.
### Summary

```c
#include <memory.h>  // Required only for function declarations
#include <string.h>   // Use either string.h or memory.h
```

```c
int memcmp (buf1, buf2, cnt);
```

- `buf1`: First buffer
- `buf2`: Second buffer
- `cnt`: Number of characters

### Description

The **`memcmp`** function does unsigned byte comparisons of the first `cnt` bytes of `buf1` and `buf2` and returns a value indicating their relationship, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>buf1</code> less than <code>buf2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>buf1</code> identical to <code>buf2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>buf1</code> greater than <code>buf2</code></td>
</tr>
</tbody>
</table>

### Return Value

The **`memcmp`** function returns an integer value, as described above.

### See Also

`memccpy, memchr, memcpy, memset`

### Example

```c
#include <string.h>

char first[100], second[100];
int result;

main()
{
    strcpy(first, "12345678901234567890");
}
memcmp

```c
strcpy(second, "12345678901234567891");
result = memcmp(first, second, 100);
printf("First is %s second.\n", 
(result < 0 ? "less than" : "greater than") : "equal to");
```

This program uses `memcmp` to compare the strings named `first` and `second`. If the first 100 bytes of the strings are equal, the program considers the strings to be equal.
## Summary

```c
#include <memory.h>
#include <string.h>
```

- Required only for function declarations
- Use either `string.h` or `memory.h`

```c
char *memcpy(dest, src, cnt);
```

- Pointer to destination
- Pointer to source
- Number of characters

## Description

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 being overwritten.

## Return Value

The `memcpy` function returns a pointer to `dest`.

## See Also

`memccpy`, `memchr`, `memcmp`, `memset`

## Example

```c
#include <memory.h>

char source[200], dest[200];
char *result;

main()
{
    strcpy(source, "This is the source to be moved."");
    strcpy(dest, ".............................");

    /* Move 200 bytes from source to dest. */
    /* and return a pointer to dest.: */
    printf("source = %s\n\ndestination = %s\n\n", source, dest);
    result = memcpy(dest, source, 200);
    printf("source = %s\n\ndestination = %s\result = %s\n", 
           source, dest, result);
}  
```
This program uses \texttt{memcpy} to copy 200 bytes from source to destination and returns a pointer to destination.
Summary

#include <memory.h>
#include <string.h>

int memicmp (buf1, buf2, cnt);
char *buf1;
char *buf2;
unsigned cnt;

Description

The `memicmp` function compares the first `cnt` bytes of `buf1` and `buf2` byte-by-byte, without regard to the case of letters in the two buffers; that is, uppercase (capital) and lowercase letters are considered equivalent. All uppercase letters in `buf1` and `buf2` are converted to lowercase before the comparison. The `memicmp` function returns a value indicating the relationship of `buf1` and `buf2`, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>buf1</code> less than <code>buf2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>buf1</code> identical to <code>buf2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>buf1</code> greater than <code>buf2</code></td>
</tr>
</tbody>
</table>

Return Value

The `memicmp` function returns an integer value, as described above.

See Also

`memccpy`, `memchr`, `memcmp`, `memcpy`, `memset`
memicmp

Example

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

char first[100], second[100];
int result;

main()
{
    strcpy(first, "Those Who Will Not Learn from History");
    strcpy(second,"THOSE WHO WILL NOT LEARN FROM their mistakes");
    /* Note that the 29th letter is right here */

    result = memicmp(first,second,29);  /* result is 0 */
    printf("%d\n",result);
}

Output:

0

This program uses memicmp to compare the the first 29 letters of the strings named first and second without regard to the case of the letters.
- memmax

- Summary

#include <malloc.h>

unsigned int __memmax();

- Description

The __memmax function returns the size (in bytes) of the largest contiguous block of memory that can be allocated from the near heap.

- Return Values

The function returns the block size, if successful.

- See Also

- Example
memmove

- Summary

```c
#include <string.h>

void *memmove(s1, s2, n);
```

void *s1;  // Target object
const void *s2;  // Source object
size_t n;  // Number of characters to copy

- Description

The `memmove` function copies `n` characters from the source object `s2` to the target object `s1`. Copying between objects that overlap is done correctly.

- Return Value

The `memmove` function returns the value of `s1`.

- See Also

- Example
**Summary**

```c
#include <memory.h>  // Required only for function declarations
#include <string.h>  // Use either string.h or memory.h

void *memset(dest, c, cnt);
void *dest;           // Pointer to destination
int c;               // Character to set
size_t cnt;          // Number of characters
```

**Description**

The `memset` function sets the first `cnt` bytes of `dest` to the character `c`.

**Return Value**

The `memset` function returns a pointer to `dest`.

**See Also**

`memccpy`, `memchr`, `memcpy`, `memcmp`

**Example**

```c
#include <memory.h>

char buffer[100];

main()
{
    char *result;

    /* Set first 20 bytes of "buffer" to 'X': */
    result = memset(buffer, 'X', 20);

    /* Tag on an end-of-string character: */
    buffer[20] = '\0';

    printf("Buffer = \%s", buffer);
}
```
memset

This program uses \texttt{memset} to set the first 20 bytes of \texttt{buffer} to \texttt{x}. It then appends a null character (\texttt{\0}) to the buffer and displays it.
Summary

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

int mkdir(pathname);
char *pathname;  Path name for new directory

Description

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.

Return Value

The mkdir function returns the value 0 if the new directory was created. A return value of -1 indicates an error, and errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Directory not created. The given name is the name of an existing file, directory, or device.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Path name not found.</td>
</tr>
</tbody>
</table>

See Also

chdir, rmdir

Example

#include <direc.h>

main()
{
  int result;
  /* "b:\tmp" could also be used in this call: */
  result = mkdir("b:/tmp");
  if (result == 0)
    printf("Directory 'b:/tmp' was successfully created\n");
mkdir

else
    printf("Problem creating directory 'b:/tmp'\n");

/* "tmp\sub" could also be used: */
result = mkdir("tmp/sub");
if (result == 0)
    printf("Directory 'tmp/sub' was successfully created\n");
else
    printf("Problem creating directory 'tmp/sub'\n");
}

This program uses mkdir to create the directories b:/tmp and tmp/sub.
# Summary

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

char *mktemp(char *template); // File-name pattern
```

## Description

The `mktemp` function creates a unique file name by modifying the given `template`. The `template` argument has the form:

```
baseXXXXXX
```

where `base` is the part of the new file name supplied by the user and the Xs are placeholders for the part supplied by `mktemp`; `mktemp` preserves `base` and replaces the six trailing Xs with an alphanumeric character followed by a five-digit value. The five-digit value is a unique number identifying the calling process. The alphanumeric character is 0 ('0') the first time `mktemp` is called with a given `template`.

In subsequent calls from the same process with the same `template`, `mktemp` checks to see if previously returned names have been used to create files. If no file exists for a given name, `mktemp` returns that name. If files exist for all previously returned names, `mktemp` creates a new name by replacing the alphanumeric character in the name with the next available lowercase letter. For example, if the first name returned is `t012345` and this name is used to create a file, the next name returned will be `ta12345`. When creating new names, `mktemp` uses, in order, '0' and the lowercase letters 'a' to 'z'.

## Return Value

The `mktemp` function returns a pointer to the modified `template`. The return value is `NULL` if the `template` argument is badly formed or no more unique names can be created from the given `template`. 
**mktemp**

- See Also

`fopen`, `getpid`, `open`

---

**Note**

The **mktemp** function generates unique file names but does not create or open files.

---

- **Example**

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

char *template = "fnXXXXXX";
char *result;

char names[5][9];

main()
{
    int i;

    for ( i = 0; i < 5; i++)
    {
        strcpy(names[i], template);

        /* Attempt to find a unique file name: */
        result = mktemp(names[i]);

        if (result == NULL)
            printf("Problem creating the template\n");
        else {
            printf("Unique file name is %s\n", result);
            fopen(result, "w");
        }
    }
}
```

The following program uses **mktemp** to create five unique file names. It opens each file name to ensure that the next name is unique.
# Summary

```c
#include <time.h>

time_t mktime(timeptr);
struct tm *timeptr;                Local time structure
```

## Description

The `mktime` function converts the time in the structure that `timeptr` points to (expressed as local time), into a calendar value. The converted time has the same encoding as the values returned by the `time` function. The original values of the `tm_wday` and `tm_yday` components of the `timeptr` structure are ignored, and the original values of the other components are not restricted to their normal ranges, which appear in `time.h`.

If successful, `mktime` sets the values of `tm_wday` and `tm_yday` appropriately, and sets the other components to represent the specified calendar time, but with their values forced to the normal ranges; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

## Return Values

The `mktime` function returns the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t)-1`.

## See Also

## Example

```c

* but must be non-negative.
```

351
modf

■ Summary

#include <math.h>

double modf(x, intptr);
double x;            Floating-point value
double *intptr;     Pointer to stored integer portion

■ Description

The modf function breaks down the floating-point value x into fractional and integer parts. The signed fractional portion of x is returned. The integer portion is stored as a floating-point value at intptr.

■ Return Value

The modf function returns the signed fractional portion of x. There is no error return.

■ See Also

frexp, ldexp

■ Example

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

main()
{
    double x, y, n;
    x = -14.87654321; /* Divide x into its fractional */
    y = modf(x, &n); /* and integer parts */
    printf("The modf(\%f, \&n) = \%f and n = \%f", x, y, n);
}
This program uses \texttt{modf} to divide the floating-point value $-14.87654321$ into its fractional and integer parts.
movedata

Summary

```c
#include <memory.h>  // Required only for function declarations
#include <string.h>  // Use either string.h or memory.h

void movedata(srcseg, srcoff, destseg, destoff, nbytes);
unsigned int srcseg;  // Segment address of source
unsigned int srcoff;  // Segment offset of source
unsigned int destseg; // Segment address of destination
unsigned int destoff; // Segment offset of destination
unsigned nbytes;      // Number of bytes
```

Description

The `movedata` function copies `nbytes` bytes from the source address specified by `srcseg:srcoff` to the destination address specified by `destseg:destoff`.

The `movedata` function is intended to be used to move far data in small- or medium-model programs where segment addresses of data are not implicitly known. In large-model programs, the `memcpy` function can be used since segment addresses are implicitly known.

Return Value

There is no return value.

See Also

`memcpy`, `segread`, `FP_SEG`
Note

Segment values for the srcseg and destseg arguments can be obtained by using either the segread function or the FP_SEG macro.

The movedata function does not handle all cases of overlapping moves correctly (overlapping moves occur when part of the destination is the same memory area as part of the source). Overlapping moves are handled correctly in the memcpy function.

Example

```c
#include <memory.h>
#include <dos.h>
#include <malloc.h>

char far *src;
char far *dest;

main()
{
    src = _fmalloc(512);
    dest = _fmalloc(512);

    movedata(FP_SEG(src), FP_OFF(src), FP_SEG(dest),
             FP_OFF(dest), 512);
    printf("The data have been moved\n");
}
```

This program uses movedata to move 512 bytes of data from src to dest.
_msize

**Summary**

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

Required only for function declarations

```c
unsigned _msize(ptr);
char *ptr;
```

Pointer to memory block

**Description**

The _msize function returns the size, in bytes, of the memory block allocated by a call to `calloc`, `malloc`, or `realloc`.

**Return Value**

The size in bytes is returned as an unsigned integer.

**See Also**

`calloc`, `_expand`, `malloc`, `realloc`
Example

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

main()
{
    long *oldptr;
    unsigned int newsize = 64000;

    oldptr = (long *)malloc(10000*sizeof(long));

    /* Get size of original memory: */
    printf("Size of memory block pointed to by oldptr = %u\n",
        _msize(oldptr));

    if (_expand(oldptr,newsize) != NULL)
        /* if _expand succeeded: */
        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));
}

Sample output:

Size of memory block pointed to by oldptr = 40000
expand was able to increase block to only 44718

This program allocates a block of memory for oldptr and then uses _msize to display the size of that block. Next, it uses expand to expand the amount of memory used by oldptr and then calls _msize again to display the new amount of memory allocated to oldptr.
### Summary

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

void _nfree(ptr);
char near *ptr;                       Pointer to allocated memory block
```

### Description

The `_nfree` function deallocates a memory block. The argument `ptr` points to a memory block previously allocated through a call to `_nmalloc`. The number of bytes freed is the number of bytes specified when the block was allocated. After the call, the freed block is again available for allocation.

### Return Value

There is no return value.

### See Also

`_nmalloc`, `free`, `malloc`

---

**Note**

Attempting to free an invalid `ptr` (a pointer not allocated with `_nmalloc`) may affect subsequent allocation and cause errors.
Example

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

char near *alloc;

main()
{
    /* Allocate memory block; then */
    /* test for a valid pointer: */

    if ((alloc = _nmalloc(100)) == NULL)
        printf("Unable to allocate memory\n");
    else
        printf("Memory sucessfully allocated");

    /* Free allocated memory: */
    _nfree(alloc);
}
```

This program uses `-nmalloc` to allocate a block of memory large enough for 100 characters inside the default data segment. Then it uses `-nfree` to return this memory to the heap.
• Summary

#include <malloc.h>

int _nheapchk()

• Description

The _nheapchk routine performs a minimal consistency check on the near heap.

• Return Value

The function returns a 0 if successful or a non-zero value otherwise.

• See Laos

• Example
_nheapset

■ Summary

#include <malloc.h>

int _nheapset(fill);
int fill; Fill character

■ Description

The _nheapset routine sets the contents of the near heap’s free nodes to
the specified fill character.

■ Return Values

The function returns a 0 if successful and a non-zero value otherwise,
corresponding to one of the following manifest constants (defined in
malloc.h):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_HEAPEMPTY</td>
<td>The near heap isn’t initialized.</td>
</tr>
<tr>
<td>_HEAPBADBEGIN</td>
<td>Can’t find the heap.</td>
</tr>
<tr>
<td>_HEAPBADNODE</td>
<td>Bad node or the near heap is damaged.</td>
</tr>
</tbody>
</table>

■ See Also

_nheapchk, _nheapwalk, _fheapchk, _fheaset, _fheapwalk

■ Example
\_nheapwalk

- Summary

\# include <malloc.h>

int \_nheapwalk(&entry);

\_heapinfo entry; Heap entry data structure

- Description

The \_nheapwalk routine returns the address of the next near heap entry data structure. If entry is passed as a NULL, the function returns information about the first entry in the heap.

The data structure contains a pointer to the heap entry, the size of the heap entry, and the free/in-use flag. The structure is defined in malloc.h.

- Return Value

- See Also

- Example
_nmalloc

■ Summary

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

char near *_nmalloc(size); Bytes in allocated block

■ Description

The _nmalloc function allocates a memory block of at least size bytes inside the default data segment. (The block may be larger than size bytes due to space required for alignment.)

■ Return Value

The _nmalloc function returns a near pointer to a char. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than char, use a type cast on the return value. The return value is NULL if there is insufficient memory available.

■ See Also

_nfree, _nmsize, malloc, realloc

■ Example

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

int near *intarray;

main()
{
  /* Allocate storage & test for a valid pointer */
  if ((intarray = (int near *)_nmalloc(20*sizeof(int))) == NULL),
    printf("Unable to allocate memory\n");
  else
    printf("Memory sucessfully allocated");
}
This program uses \texttt{\_nmalloc} to allocate a block of memory large enough for 20 integers within the default data segment.
_nmsize

■ Summary

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

unsigned _nmsize(ptr);
char near *ptr;  
Pointer to memory block

■ Description

The _nmsize function returns the size in bytes of the memory block allocated by a call to _nmalloc.

■ Return Value

The _nmsize function returns the size in bytes as an unsigned integer.

■ See Also

_ffree, _fmalloc, _fmsize, malloc, _msize, _nfree, _nmalloc

■ Example

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

main()
{
    char near *stringarray;

    stringarray = _nmalloc(200*sizeof(char));

    if (stringarray != NULL) /* _nmalloc" succeeded. */
        printf("%u bytes allocated\n","nmsize(stringarray));
    else /* _nmalloc" failed */
        printf("Allocation request failed.\n");
}

This program uses _nmsize to get the number of bytes in the allocated block, then displays this size.
 Summary

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
```

Required only for function declarations

```c
int open(char *pathname, int oflag [,int pmode]);
```

File path name

```c
char *pathname;
```

Type of operations allowed

```c
int oflag;
```

Permission setting

### Description

The `open` function opens the file specified by `pathname` and prepares the file for subsequent reading or writing, as defined by `oflag`. The argument `oflag` is an integer expression formed by combining one or more of the following manifest constants, defined in `fcntl.h`. When more than one manifest constant is given, the constants are joined with the bitwise-OR operator (`|`).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_APPEND</td>
<td>Reposition the file pointer to the end of the file before every write operation.</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>Create and open a new file for writing; this has no effect if the file specified by <code>pathname</code> exists.</td>
</tr>
<tr>
<td>O_EXCL</td>
<td>Return an error value if the file specified by <code>pathname</code> exists. Only applies when used with <code>O_CREAT</code>.</td>
</tr>
<tr>
<td>O_RDONLY</td>
<td>Open file for reading only; if this flag is given, neither <code>O_RDWR</code> nor <code>O_WRONLY</code> can be given.</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>Open file for both reading and writing; if this flag is given, neither <code>O_RDONLY</code> nor <code>O_WRONLY</code> can be given.</td>
</tr>
<tr>
<td>O_TRUNC</td>
<td>Open and truncate an existing file to zero length; the file must have write permission. The contents of the file are destroyed.</td>
</tr>
</tbody>
</table>
open

O_ WONLY
Open file for writing only; if this flag is given, neither O_RDONLY nor O_RDWR can be given.

O_BINARY
Open file in binary (untranslated) mode. (See fopen for a description of binary mode.)

O_TEXT
Open file in text (translated) mode. (See fopen for a description of text mode.)

Note
Use the O_TRUNC flag with care, as it destroys the complete contents of an existing file.

Either O_RDONLY, O_RDWR, or O_WRONLY must be given to specify the access mode. The access mode does not default.

The pmode argument is required only when O_CREAT is specified. If the file exists, pmode is ignored. Otherwise, pmode specifies the file’s permission settings, which are set when the new file is closed for the first time. The pmode is an integer expression containing one or both of the manifest constants S_IWRITE and S_IREAD, defined in sys/stat.h. When both constants are given, they are joined with the bitwise-OR operator (|). The meaning of the pmode argument is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IWRITE</td>
<td>Writing permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>Reading permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IWRITE</td>
</tr>
</tbody>
</table>

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.

Important
Under DOS Version 3.x with SHARE installed, a bug occurs when opening a new file with oflag set to O_CREAT | O_RDONLY or O_CREAT | O_WRONLY with pmode set to S_IREAD. In this
case, the operating system will prematurely close the file during system calls made within `open`.

To get around the problem, open the file with `flpmode` set to `S_IWWRITE`. After closing the file, call `chmod` and change the mode back to `S_IRREAD`. Another work-around is to open the file with `pmod` set to `S_IRREAD` and `omode` set to `O_CREAT | O_RDWR`.

---

The `open` function applies the current file-permission mask to `pmod` before setting the permissions (see `umask`).

- **Return Value**

  The `open` function returns a file handle for the opened file. A return value of -1 indicates an error, and `errno` is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Given path name is a directory; or an attempt was made to open a read-only file for writing; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; DOS Version 3.0 or later only).</td>
</tr>
<tr>
<td>EEXIST</td>
<td>The <code>O_CREAT</code> and <code>O_EXCL</code> flags are specified but the named file already exists.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>No more file handles available (too many open files).</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found.</td>
</tr>
</tbody>
</table>

- **See Also**

  `access`, `chmod`, `close`, `creat`, `dup`, `dup2`, `fopen`, `sopen`, `umask`
open

- Example

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

main()
{
  int fh1, fh2;

  fh1 = open("data1", O_RDONLY);
  if (fh1 == -1)
    perror("open failed on input file");
  else
    printf("open succeeded on input file\n");

  fh2 = open("data2", O_WRONLY|O_CREAT, S_IREAD|S_IWRITE);
  if (fh2 == -1)
    perror("open failed on output file");
  else
    printf("open succeeded on output file\n");
}
```

This program uses `open` to open a file named `data1` for input and a file named `data2` for output.
# Summary

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

int outp(port, value);
unsigned port; // Port number
int value; // Output value
```

## Description

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 0 to 65,535; `value` can be any integer in the range 0 to 255.

## Return Value

The `outp` function returns `value`. There is no error return.

## See Also

`inp`

## Example

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

int port, byte_val;

main()
{
    port = 1;
    byte_val = 3;
    outp(port, byte_val);
    printf("The value %d has been output to port %d",
           byte_val, port);
}
```

This program uses `outp` to write the value 3 to output port 1.
**perror**

- **Summary**

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

void perror(string);
const char *string;
int errno;
int sys_nerr;
char *sys_errlist[sys_nerr];
```

**Description**

The `perror` function prints an error message to `stderr`. The `string` argument is printed first, followed by a colon, the system error message for the last library call that produced an error, and a new line. If `string` is a `NULL` pointer or a pointer to a `NULL` string, `perror` prints only the system error message.

The actual error number is stored in the variable `errno`, which should be declared at the external level. The system error messages are accessed through the variable `sys_errlist`, which is an array of messages ordered by error number. The `perror` function prints the appropriate error message by using the `errno` value as an index to `sys_errlist`. The value of the variable `sys_nerr` is defined as the maximum number of elements in the `sys_errlist` array.

To produce accurate results, `perror` should be called immediately after a library routine returns with an error. Otherwise, the `errno` value may be overwritten by subsequent calls.

- **Return Value**

The `perror` function returns no value.
See Also

clearerr, ferror, strerror

Note

Under DOS, some of the errno values listed in errno.h are not used. See Appendix A, “Error Messages,” for a list of errno values used on DOS and the corresponding error messages. The perror function prints an empty string for any errno value not used under DOS.

Example

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>

main()
{
    int fh1, fh2;

    fh1 = open("data1", O_RDONLY);
    if (fh1 == -1)
        perror("open failed on input file");
    else
        printf("open succeeded on input file\n");

    fh2 = open("data2", O_WRONLY|O_CREAT, S_IWRITE);
    if (fh2 == -1)
        perror("open failed on output file");
    else
        printf("open succeeded on output file\n");
}
```

This program opens a file named data1 for input and a file named data2 for output. If either open operation fails, the program displays an error message to indicate the failure.
pow

- Summary

```c
#include <math.h>

double pow(x, y);
double x; Number to be raised
double y; Power of x
```

- Description

The `pow` function computes \( x \) raised to the \( y \)th power.

- Return Value

The `pow` function returns the value of \( x^y \). If \( x \) is not zero and \( y \) is zero, `pow` returns the value 1. If \( x \) is 0 and \( y \) is negative, `pow` sets `errno` to `EDOM` and returns `HUGE_VAL`. If both \( x \) and \( y \) are 0, or if \( x \) is negative and \( y \) is not an integer, the function prints a `DOMAIN` error message to `stderr`, sets `errno` to `EDOM`, and returns 0. If an overflow results, the function sets `errno` to `ERANGE` and returns 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 than \( 2^{64} \). For example, \( 1.0E100 \) is not recognized as an integral value.

- See Also

`exp, log, sqrt`

- Example

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

main()
{
    double x = 2.0, y = 3.0, z;
    z = pow(x, y); /* z is 8.0 \( 2^3 \) */
    printf("The pow(\%.2f,\%.2f) = \%.2f\", x, y, z);
```
This program uses \texttt{pow} to calculate the value of $2^3$. 
printf

- Summary

#include <stdio.h>

int printf(format-string[, argument...]);
const char *format-string;

- Description

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 (if arguments follow the `format-string`) format specifications. Ordinary characters and escape sequences are simply copied to `stdout` in order of their appearance. For example, the line

```c
printf("Line one\n\t\tLine two\n");
```

produces the output

Line one

        Line two

(For more information on escape sequences, see Section 2.2.4, “Escape Sequences,” in the Microsoft C Optimizing Compiler Language Reference.)

If `arguments` follow the `format-string`, then the `format-string` must contain format specifications that determine the output format for these `arguments`. Format specifications always begin with a percent sign (%), and are described in greater detail below.

The `format-string` is read left to right. When the first format specification (if any) is encountered, the value of the first `argument` after the `format-string` is converted and output according to the format specification. The second format specification causes the second `argument` to be converted and output, and this continues 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}]]type
```
Each field of the format specification is a single character or a number signifying a particular format option. The \textit{type} character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, or a number (see Table R.1). The simplest format specification contains only the percent sign and a \textit{type} character (for example, \texttt{%s}). The optional fields control other aspects of the formatting, as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>Justification of output and printing of signs, blanks, decimal points, octal and hexadecimal prefixes (see Table R.2).</td>
</tr>
<tr>
<td>width</td>
<td>Minimum number of characters output.</td>
</tr>
<tr>
<td>precision</td>
<td>Maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values (see Table R.3).</td>
</tr>
<tr>
<td>F, N</td>
<td>Prefixes that allow user to override default addressing conventions of memory model being used:</td>
</tr>
<tr>
<td>F</td>
<td>Used in small model to print value that has been declared \texttt{far}</td>
</tr>
<tr>
<td>N</td>
<td>Used in medium, large and huge models for \texttt{near} value</td>
</tr>
<tr>
<td>F and N should be used only with the \texttt{s} and \texttt{p} \textit{type} characters, since they are relevant only with arguments that pass a pointer.</td>
<td></td>
</tr>
<tr>
<td>h, l</td>
<td>Size of argument expected:</td>
</tr>
<tr>
<td>h</td>
<td>Used as a prefix with the integer types \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x}, and \texttt{X} to specify that the argument is a \texttt{short int}</td>
</tr>
<tr>
<td>l</td>
<td>Used as a prefix with \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x}, and \texttt{X} types to specify that the argument is a \texttt{long int}; also used as a prefix with \texttt{e}, \texttt{E}, \texttt{f}, \texttt{g}, and \texttt{G} types to show that the argument is \texttt{double}, rather than \texttt{float}</td>
</tr>
</tbody>
</table>

If a percent sign (\%) is followed by a character that has no meaning as a format field, the character is simply copied to \texttt{stdout}. For example, to print a percent-sign character, use \texttt{\%\%).
### printf

#### Table R.1

printf Type Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument Type</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Integer</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>i</td>
<td>Integer</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>u</td>
<td>Integer</td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td>o</td>
<td>Integer</td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td>x</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using “abcdef”</td>
</tr>
<tr>
<td>X</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using “ABCDEF”</td>
</tr>
<tr>
<td>f</td>
<td>Floating point</td>
<td>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.</td>
</tr>
<tr>
<td>e</td>
<td>Floating point</td>
<td>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 –</td>
</tr>
<tr>
<td>E</td>
<td>Floating point</td>
<td>Identical to the “e” format, except that “E” introduces the exponent instead of “e”</td>
</tr>
<tr>
<td>g</td>
<td>Floating point</td>
<td>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.</td>
</tr>
<tr>
<td>G</td>
<td>Floating point</td>
<td>Identical to the “g” format, except that “E” introduces the exponent (where appropriate) instead of “e”</td>
</tr>
<tr>
<td>c</td>
<td>Character</td>
<td>Single character</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Characters printed up to the first null character (’\0’) or until precision is reached</td>
</tr>
</tbody>
</table>
Table R.1 (continued)

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument Type</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Pointer to integer</td>
<td>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.</td>
</tr>
<tr>
<td>p</td>
<td>Far pointer</td>
<td>Prints the address pointed to by the argument in the form \texttt{xxxx:yyyy}, where \texttt{xxxx} is the segment and \texttt{yyyy} is the offset, and the digits \texttt{x} and \texttt{y} are uppercase hexadecimal digits; \texttt{%Np} prints only the offset of the address, \texttt{yyyy}. Since \texttt{%p} expects a pointer to a far value, pointer arguments to \texttt{p} must be cast to \texttt{far} in small-model programs.</td>
</tr>
</tbody>
</table>

Table R.2

printf Flag Characters

<table>
<thead>
<tr>
<th>Flag$^1$</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>Left justify the result within the field \textit{width}.</td>
<td>Right justify</td>
</tr>
<tr>
<td>+</td>
<td>Prefix the output value with a sign (+ or –) if the output value is of a signed type.</td>
<td>Sign appears only for negative signed values (–).</td>
</tr>
<tr>
<td>blank (′ ′)</td>
<td>Prefix the output value with a blank if the output value is signed and positive; the \textit{blank} is ignored if both the \textit{blank} and “+” flags appear.</td>
<td>No blank</td>
</tr>
<tr>
<td>#</td>
<td>When used with the \texttt{o}, \texttt{x}, or \texttt{X} format, the “#” flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.</td>
<td>No prefix</td>
</tr>
</tbody>
</table>
### printf

**Table R.2 (continued)**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>When used with the e, E, or f format, the &quot;#&quot; flag forces the output value to contain a decimal point in all cases.</td>
<td>Decimal point appears only if digits follow it.</td>
</tr>
<tr>
<td></td>
<td>When used with the g or G format, the &quot;#&quot; flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.</td>
<td>Decimal point appears only if digits follow it. Trailing zeros are truncated.</td>
</tr>
<tr>
<td></td>
<td>Ignored when used with c, d, i, u, or s</td>
<td></td>
</tr>
</tbody>
</table>

1 More than one flag can appear in a format specification.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or not-a-number, the `printf` function gives the following output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ infinity</td>
<td>1.# INF random-digits</td>
</tr>
<tr>
<td>- infinity</td>
<td>-1.# INF random-digits</td>
</tr>
<tr>
<td>indefinite</td>
<td>digit.# IND random-digits</td>
</tr>
<tr>
<td>not-a-number</td>
<td>digit.# NAN random-digits</td>
</tr>
</tbody>
</table>

The *width* is a non-negative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added 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 0, zeros are added until the minimum width is reached (not useful for left-justified numbers).

The *width* specification never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

The *width* specification may be an asterisk (*), in which case an int argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. In no case does a nonexistent or small field width cause a truncation of a field; if the result
of a conversion is wider than the field width, the field is expanded to contain the conversion result.

The \textit{precision} specification is a non-negative decimal integer preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits (see Table R.3). Unlike the \textit{width} specification, the \textit{precision} can cause truncation of the output value, or rounding in the case of a floating-point value.

The \textit{precision} specification may be an asterisk (*), in which case an \texttt{int} argument from the argument list supplies the value. The \textit{precision} argument must precede the value being formatted in the argument list.

The interpretation of the \textit{precision} value, and the default when \textit{precision} is omitted, depend on the \textit{type}, as shown in Table R.3.

\begin{table}
\centering
\caption{How printf Precision Values Affect Type}
\begin{tabular}{lll}
\hline
Type & Meaning & Default \\
\hline
d & The \textit{precision} specifies the minimum number of digits to be printed. If the & If \textit{precision} is 0 or omitted entirely, or if the period (.) appears without a number following it, the \textit{precision} is set to 1. \\
i & number of digits in the argument is less than \textit{precision}, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds \textit{precision}. & \\
u & & Default \textit{precision} is six; if \textit{precision} is zero or the period (.) appears without a number following it, no decimal point is printed. \\
o & & Default \textit{precision} is zero; if \textit{precision} is explicitly zero, no decimal point appears. \\
x & & All significant digits are printed. \\
X & & Character printed \\
e & The \textit{precision} specifies the number of digits to be printed after the decimal point. The last printed digit is rounded. & \\
E & & \\
f & The \textit{precision} value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. & \\
f & & \\
g & The \textit{precision} specifies the maximum number of significant digits printed. & \\
G & & \\
c & No effect & \\
\hline
\end{tabular}
\end{table}
printf

Table R.3 (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>The precision specifies the maximum number of characters to be printed. Characters in excess of precision are not printed.</td>
<td>Characters are printed until a null character is encountered.</td>
</tr>
</tbody>
</table>

- **Return Value**

The `printf` function returns the number of characters printed.

- **See Also**

`fprintf`, `scanf`, `sprintf`, `vfprintf`, `vprintf`, `vsscanf`

- **Example**

```c
main() /* Format and print various data. */
{
    char ch = 'h', *string = "computer";
    int count = 234, *ptr, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;

    printf("%d %+d %06d %X %x %o\n\n", count, count, count, count);

    printf("1234567890123%n45678901234567890\n\n", &count);
    printf("Value of count should be 13; count = %d\n\n", count);

    printf("%10c%5c\n\n", ch, ch);
    printf("%25s\n%25.4s\n\n", string, string);
    printf("%f %.2f %e %E\n\n", fp, fp, fp, fp);
    printf("%i %i %i\n\n", hex, oct, dec);

    ptr = &count;
    printf("%Np %p %Fp\n",
            ptr, (int far *) ptr, (int far *) ptr);
}`
```

382
Output:

```
234  +234  000234  EA   ea  352
123456789012345678901234567890
Value of count should be 13; count = 13
    h    h
  computer
    comp
251.736600  251.74  2.517366e+002  2.517366E+002
16   8    10
127A   1328:127A  1328:127A
```

This program uses `printf` to display various strings, numbers, characters, and pointers with different formats.
putc – putchar

- **Summary**

```c
#include <stdio.h>

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

int putchar(c);
int c;
```

- **Description**

The `putc` routine writes the single character `c` to the output `stream` at the current position. The `putchar` routine is identical to `putc(c, stdout)`.

- **Return Value**

The `putc` and `putchar` routines return the character written. A return value of `EOF` indicates an error, which could be caused by an attempt to write to a read-only file, specifying an invalid stream pointer, or no space left on the device. Since the `EOF` value is a legitimate integer value, the `ferror` function should be used to verify that an error occurred.

- **See Also**

fputc, fputchar, getc, getchar

---

**Note**

The `putc` and `putchar` routines are identical to `fputc` and `fputchar`, but are macros, not functions.
Example

```c
#include <stdio.h>

FILE *stream;
char buffer[81] = "This is the line of output\n";
int i, ch;

main()
{
    stream = stdout;

    /* Write line to the stream: */
    for (i = 0; (i < 81) &&
         ((ch = putc(buffer[i],stream)) != EOF); i++)
    }

    /* Note that the body of the "for" statement is null, since the
     ** write operation is carried out in the test expression.
     */

This program uses **putc** to write **buffer** to a stream. If an error occurs,
the program will stop before writing the entire buffer.
```
putch

• Summary

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

void putch(c)
int c;  
Character to be output

• Description

The putch function writes the character c directly to the console.

• Return Value

There is no return value.

• See Also

cprintf, getch, getche

• Example

#include <conio.h>

mygetche() {
    int ch;
    ch = getch();  /* Get a character with no echo */
    putch(ch);  /* Send the character to the console */
    return(ch);
}

main()
{
    char ch;
    while (( ch = mygetche() ) != '\r');
}

This program uses getch and putch to implement a function similar to getche.
putenv

- Summary

```c
#include <stdlib.h>

int putenv(envstring);
char *envstring;
```

Required only for function declarations

Environment string definition

- Description

The `putenv` function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program).

The `envstring` argument must be a pointer to a string with the form

```
varname=string
```

where `varname` is the name of the environment variable to be added or modified and `string` is the variable’s value. If `varname` is already part of the environment, it is replaced by `string`; otherwise, the new `string` is added to the environment. A variable can be set to an empty value by specifying an empty `string`.

This function affects only the environment that is local to the currently running process; it cannot be used to enter new items in the command-level environment. When the currently running process terminates, the environment reverts to the level of the parent process (in most cases, the DOS level). However, the environment affected by `putenv` is passed to any child processes created by `spawn` or `exec`, and these child processes set any new items added by `putenv`.

Never free a pointer to an environment entry or the environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to `putenv`, then exit the function in which the variable is declared.

---

**Note**

Environment table entries must not be changed directly. If an entry must be changed, use `putenv`. To modify the returned value without affecting the environment table, use `strdup` or `strcpy` to make a copy.
putenv

of the string.

The `getenv` and `putenv` functions use the global variable `environ` to access the environment table. The `putenv` function may change the value of `environ`, thus invalidating the `envp` argument to the `main` function. Therefore, it's safer to use the `environ` variable to access the environment information.

- **Return Value**

The `putenv` function returns 0 if it is successful. A return value of -1 indicates an error.

- **See Also**

`getenv`

Note

The `getenv` and `putenv` functions use the global variable `environ` to access the environment table. The `putenv` function may change the value of `environ`, thus invalidating the "envp" argument to the "main" function.

- **Example**

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

main()
{
    /* Attempt to change directory path: */
    if (putenv("PATH=a\bin;b:\tmp") == -1)
    {
        printf("putenv failed -- out of memory");
        exit(1);
    }
```
else
    printf("'putenv' worked.\n");
}

This program uses **putenv** to change the value of the **PATH** variable in the environment table.
puts

■ Summary

#include <stdio.h>

int puts(string);
const char *string; String to be output

■ Description

The puts function writes the given string to the standard output stream stdout, replacing the string's terminating null character ('\0') with a new-line character ('\n') in the output stream.

■ Return Value

The puts function returns the last character written, which is always the new-line character ('\n'). A return value of EOF indicates an error.

■ See Also

fputs, gets

■ Example

#include <stdio.h>

int result;

main()
{
  /* Write a prompt to "stdout": */
  result = puts("Insert data disk and strike any key");
}

This program usesputc to write a string to stdout.
## Summary

```c
#include <stdio.h>

int putw(binint, stream);

int binint;    // Binary integer to be output
FILE *stream;  // Pointer to file structure
```

## Description

The `putw` function writes a binary value of type `int` to the current position of the specified `stream`. The `putw` function does not affect the alignment of items in the stream, nor does it assume any special alignment.

## Return Value

The `putw` function returns the value written. A return value of `EOF` may indicate an error. Since `EOF` is also a legitimate integer value, `ferror` should be used to verify an error.

## See Also

`getw`

---

**Note**

The `putw` function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with `putw`, since the size of an `int` and ordering of bytes within an `int` differ across systems.
putw

**Example**

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

FILE *stream;

main()
{
    stream = fopen("data.bin", "wb");

    putw(0345, stream);  /* Write word to the stream: */

    if (ferror(stream))  /* Make an error check: */
    {
        printf("Putw failed");
        clearerr(stream);
    }
    else
        printf( "\nPutw wrote a word.\n" );
}
```

This program uses putw to write a word to a stream and then performs and error check.
qsort

Summary

#include <search.h>          // Required only for function declarations

void qsort(base, num, width, compare);
void *base;
size_t num;
size_t width;
int (*compare)(const void *elem1, const void *elem2);

Description

The qsort function implements a quick-sort algorithm to sort an array of num elements, each of width bytes. The argument base is a pointer to the base of the array to be sorted. The qsort function overwrites this array with the sorted elements.

The argument compare is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The qsort function will call 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>element1 less than element2</td>
</tr>
<tr>
<td>0</td>
<td>element1 equivalent to element2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>element1 greater than element2</td>
</tr>
</tbody>
</table>

The sorted array is in increasing order, as defined by the compare function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the compare function.

Return Value

There is no return value.
See Also

bsearch, lsearch

Example

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

int compare();   /* must declare as a function */
                /* for qsort's compare */

main (argc, argv)
int argc;
char **argv;
{
    char **result;
    int i;

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

    /* Sort remaining args using Quick sort algorithm: */
    qsort((void *)argv, (size_t)argc, sizeof(char *), compare);

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

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

This program reads the command-line parameters and uses `qsort` to sort them. It then displays the sorted arguments.
raise

- Summary

```c
#include <signal.h>

int raise(sig);
int sig;  // Signal to be sent
```

- Description

The `raise` function sends the signal `sig` to the executing program.

- Return Value

If successful, the `raise` function returns zero. Otherwise, it returns a non-zero value.

- See Also

- Example
• Summary

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

int rand();

• Description

The rand function returns a pseudo-random integer in the range 0 to 32,767. The srand routine can be used before calling rand to set a random starting point.

• Return Value

The rand function returns a pseudo-random number as described above. There is no error return.

• See Also

srand

• Example

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

main()
{
    int x;

    for (x = 1; x <= 20; x++)
        printf("Iteration %d, rand=%d\n", x, rand());
}

This program displays the first 20 random integers generated by rand.
read

- Summary

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

Required only for function declarations

```c
int read(handle, buffer, count);
```

Handle referring to open file

```c
int handle;
```

Storage location for data

```c
char *buffer;
```

Maximum number of bytes

- Description

The `read` function attempts to read `count` bytes from the file associated with `handle` into `buffer`. The read operation begins at the current position of the file pointer (if any) associated with the given file. After the read operation, the file pointer points to the next unread character.

- Return Value

The `read` function returns the number of bytes actually read, which may be less than `count` if there are fewer than `count` bytes left in the file or if the file was opened in text mode (see below). The return value 0 indicates an attempt to read at end-of-file. The return value -1 indicates an error, and `errno` is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The given <code>handle</code> is invalid; or the file is not open for reading; or the file is locked (DOS Versions 3.0 or later only).</td>
</tr>
</tbody>
</table>

If you are reading more than 32K (the maximum size for type `int`) from a file, the return value should be of type `unsigned int`. (See the example that follows.) However, the maximum number of bytes that can be read from a file is 65,534 at a time, since 65,535 (or 0xFFFF) is indistinguishable from -1, and therefore would return an error.

If the file was opened in text mode, the return value may not correspond to the number of bytes actually read. When text mode is in effect, each carriage-return–line-feed pair (CR-LF) is replaced with a single line-feed character (LF). Only the single line-feed character is counted in the return value. The replacement does not affect the file pointer.
See Also

creat, fread, open, write

Note

Under DOS, when files are opened in text mode, a character is treated as an end-of-file indicator. When the CONTROL-Z is encountered, the read terminates, and the next read returns 0 bytes. The file must be closed to clear the end-of-file indicator.

Example

```c
#include <fcntl.h>    /* Needed only for O_RDWR definition */
#include <io.h>
#include <stdio.h>

char buffer[60000];

main()
{
    int fh;
    unsigned int nbytes = 60000, bytesread;

    /* Open file for input: */
    if ((fh = open("data", 0_RDONLY)) == -1)
    {
        perror("open failed on input file");
        exit(1);
    }

    /* Read in input: */
    if ((bytesread = read(fh, buffer, nbytes)) <= 0)
    {
        perror("Problem reading file");
    }
    else
    {
        printf("Read %u bytes from file\n", bytesread);
    }
}
```

This program opens a file named data and tries to read 60,000 bytes from that file using read. It then displays the actual number of bytes read from data.
realloc

■ Summary

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

void *realloc(ptr, size);
void *ptr; Pointer to previously allocated memory block
size_t size; New size in bytes

■ Description

The realloc function changes the size of a previously allocated memory block. The ptr argument points to the beginning of the block. if ptr is null, realloc functions like malloc and allocates a new block of size bytes. 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, halloc, malloc, or realloc since the block was freed.

■ Return Value

The realloc function returns a char pointer to the reallocated memory block. The block may be moved when its size is changed; therefore, the ptr argument to realloc is not necessarily the same as the return value.

The return value is NULL if ptr is zero or if there is insufficient memory available to expand the block to the given size. The original block is freed when this occurs.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than char, use a type cast on the return value.

■ See Also

calloc, free, halloc, malloc
Example

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

char *alloc;

main()
{
    /* Get enough space for 50 characters: */
    alloc = malloc(50*sizeof(char));
    printf("Block successfully allocated\n\n");

    /* Reallocation block to hold 100 chars: */
    if (alloc != NULL)
        alloc = realloc(alloc,100*sizeof(char));

    if (alloc != NULL)
        printf("Block is sucessfully reallocated\n\n");
    else
        printf("'realloc' failed--block was freed\n\n");
}
```

This program allocates enough space for 50 characters, then uses `realloc` to expand this space so that it can hold 100 characters.
remove

- Summary

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

```c
int remove(pathname);
```

- Description

The `remove` function deletes the file specified by `pathname`.

- Return Value

The `remove` 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Path name specifies a directory or a read-only file</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found</td>
</tr>
</tbody>
</table>

- See Also

`close`, `unlink`

- Example

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

main()
{
    int result;

    result = remove("data");
    if (result == -1)
        perror("Could not delete 'data'");
    else
        printf("'data' was sucessfully deleted\n");
```
This program uses \texttt{remove} to delete a file named \texttt{data}.
rename

- Summary

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

Required only for function declarations

Use either io.h or stdio.h

```c
int rename(oldname, newname);
const char *oldname;
const char *newname;
```

Pointer to old name

Pointer to new name

- Description

The `rename` function renames the file or directory specified by `oldname` to the name given by `newname`. The `oldname` must specify the path name of an existing file or directory. The `newname` must not specify the name of an existing file or directory.

The `rename` function can be used to move a file from one directory to another by giving a different path name in the `newname` argument. However, files cannot be moved from one device to another (for example, from Drive A to Drive B). Directories can only be renamed, not moved.

- Return Value

The `rename` function returns 0 if it is successful. On an error, it returns a nonzero value and sets `errno` to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>File or directory specified by <code>newname</code> already exists or could not be created (invalid path); or <code>oldname</code> is a directory and <code>newname</code> specifies a different path.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name specified by <code>oldname</code> not found.</td>
</tr>
<tr>
<td>EXDEV</td>
<td>Attempt to move a file to a different device.</td>
</tr>
</tbody>
</table>
See Also

creat, fopen, open

Note

Note that the order of arguments in rename is the reverse of their order in pre-4.0 versions of Microsoft C. This change was made to conform to the ANSI C standard.

Example

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

main()
{
    int result;
    /* Attempt to rename file: */
    result = rename("input","data");
    /* Check for errors: */
    if (result != 0)
        perror("Was not able to rename file");
    else
        printf("File successfully renamed");
}
```

This program uses `rename` to rename a file named `input` to `data`. For this operation to succeed, a file named `input` must exist and a file named `data` must not exist.
rewind

- **Summary**

```
#include <stdio.h>

void rewind(stream);
FILE *stream;              // Pointer to file structure
```

- **Description**

The `rewind` function repositions the file pointer associated with `stream` to the beginning of the file. A call to `rewind` is equivalent to

```
void fseek(stream, 0L, SEEK_SET);
```

except that `rewind` clears the end-of-file and error indicators for the stream, and `fseek` does not; also, `fseek` returns a value that indicates whether or not the pointer was successfully moved, but `rewind` does not return any value.

- **Return Value**

There is no return value.

- **See Also**

`fseek`, `ftell`

- **Example**

```
#include <stdio.h>

FILE *stream;
int data1, data2;

main()
{
    data1 = 1;
    data2 = -37;

    /* Open for both reading and writing: */
    stream = fopen("data","w+");
```
This program first opens a file named data for input and output and writes two integers to the file. Next, it uses \texttt{rewind} to reposition the file pointer to the beginning of the file and reads the data back in.
# Summary

```c
#include <direct.h>

int rmdir(pathname);
char *pathname;
```

- **Description**

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.

- **Return Value**

The `rmdir` function returns the value 0 if the directory is successfully deleted. A return value of -1 indicates an error, and `errno` is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>The given path name is not a directory; or the directory is not empty; or the directory is the current working directory or root directory.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Path name not found.</td>
</tr>
</tbody>
</table>

- **See Also**

`chdir, mkdir`

- **Example**

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

main()
{
    int result1, result2;

    /* Remove "sub1" from root directory: */
    result1 = rmdir("/sub1");
}
```
if (result1 == -1)
    perror("Unable to remove directory");
else
    printf("Directory sucessfully removed");

/* Remove subdirectory "sub2" from */
/* the current working directory: */
result2 = rmdir("sub2");
if (result2 == -1)
    perror("Unable to remove directory");
else
    printf("Directory sucessfully removed");
}

This program uses **rmdir** to remove the subdirectory \sub1 from the root directory and the subdirectory \sub2 from the current working directory.
rmtmp

- Summary

```c
#include <stdio.h>

int rmtmp();
```

- Description

The `rmtmp` function is used to clean up all the temporary files in the current directory; `rmtmp` removes only those files created by `tmpfile`.

The `rmtmp` function should be used only in the same directory in which the temporary files were created.

- Return Value

The `rmtmp` function returns the number of temporary files closed and deleted.

- See Also

flushall, tmpfile, tmpnam

- Example

```c
#include <stdio.h>

FILE *stream;

main()
{
    int numdeleted;

    /* Create a temporary file: */
    if ((stream = tmpfile()) == NULL)
        perror("Could not open new temporary file");

    /* Remove a temporary file: */
    numdeleted = rmtmp();

    printf("Number of closed files deleted in current directory = %d\n",
           numdeleted);
}
```
This program creates a temporary file, then uses `rmtmp` to delete this file.
Summary

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

char *sbrk(incr);
int incr;                               Number of bytes added or subtracted

Description

The sbrk function resets the break value for the calling process. The break value is the address of the first byte of memory beyond the end of the default data segment. The sbrk function adds incr bytes to the break value; the size of the process’s allocated memory is adjusted accordingly. Note that incr may be negative, in which case the amount of allocated space is decreased by incr bytes.

Return Value

The sbrk function returns the old break value. A return value of (char *)–1 indicates an error, and errno is set to ENOMEM, indicating that insufficient memory was available.

See Also

calloc, free, malloc, realloc

Important

In compact-, large-, and huge-model programs, sbrk fails and returns (char *)–1. Use malloc for allocation requests in large-model programs.
sbrk

Example

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

main()
{
    char *alloc;

    alloc = sbrk(100); /* 100 bytes allocated on the heap */
    if (alloc != (char *)-1)
    {
        printf("100 bytes of memory have been allocated\n");
        printf("Now 40 bytes will be deallocated\n");
        sbrk(-40); /* Deallocate 40 bytes */
    }
    else
        perror("Problem allocating 100 bytes\n");
}

This program uses sbrk to allocate 100 bytes of memory and then to deallocate 40 bytes.
## Summary

```c
#include <stdio.h>

int scanf(format-string[, argument...]);
const char *format-string;
```

### Description

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 (blank (' '), tab ('\t'), or new line ('\n')). A white-space character causes `scanf` to read, but not store, all consecutive white-space characters in the input up to the next non-white-space character. One white-space character in the `format-string` matches any number (including 0) and combination of white-space characters in the input.

- Non-white-space characters, except for the percent-sign character (%). A non-white-space character causes `scanf` to read, but not store, a matching non-white-space character. If the next character in `stdin` does not match, `scanf` terminates.

- Format specifications, introduced by the percent sign (%). A format specification causes `scanf` to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The `format-string` is read from left to right. Characters outside format specifications are expected to match the sequence of characters in `stdin`; the matched characters in `stdin` are scanned but not stored. If a character in `stdin` conflicts with the `format-string`, `scanf` terminates. The conflicting character is left in `stdin` as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to the format specification and stored in the location specified by the first `argument`. The second format specification causes the second input field to be converted and stored in the second `argument`, and so on through the end of the `format-string`.

---

415
**scanf**

An input field is defined as all characters up to the first white-space character (space, tab, or new line), or up to the first character that cannot be converted according to the format specification, or until the field width, if specified, is reached, whichever comes first. If there are too many arguments for the given format specifications, the extra arguments are evaluated but ignored. The results are undefined if there are not enough arguments for the given format specifications.

A format specification has the following form:

```
%[*][width][{F | N}][{h | l}]type
```

Each field of the format specification is a single character or a number signifying a particular format option. The type character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a type character (for example, `%s`).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters — that is, a sequence of characters that must match the input. For example, to specify that a percent sign character is to be input, use `%%%`.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The width is a positive decimal integer controlling the maximum number of characters to be read from stdin. No more than width characters are converted and stored at the corresponding argument. Fewer than width characters may be read if a white-space character (space, tab, or new line) or a character that cannot be converted according to the given format occurs before width is reached.

The optional F and N prefixes allow the user to override the default addressing conventions of the memory model being used. F should be prefixed to an argument pointing to a far object, while N should be prefixed to an argument pointing to a near object.
The optional prefix l indicates that the long version of the following type is to be used, while the prefix h indicates that the short version is to be used. The corresponding argument should point to a long or double object (with the l character) or a short object (with the h character). The l and h modifiers can be used with the d, i, n, o, x, and u type characters. The l modifier can also be used with the e, f, and g type characters. The l and h modifiers are ignored if specified for any other type.

The type characters and their meanings are described in Table R.4.

Table 0.4
scanf Type Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Type of Input Expected</th>
<th>Type of Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Decimal integer</td>
<td>Pointer to int</td>
</tr>
<tr>
<td>D</td>
<td>Decimal integer</td>
<td>Pointer to long</td>
</tr>
<tr>
<td>o</td>
<td>Octal integer</td>
<td>Pointer to int</td>
</tr>
<tr>
<td>O</td>
<td>Octal integer</td>
<td>Pointer to long</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal integer(^1)</td>
<td>Pointer to int</td>
</tr>
<tr>
<td>X</td>
<td>Hexadecimal integer(^1)</td>
<td>Pointer to long</td>
</tr>
<tr>
<td>i</td>
<td>Decimal, hexadecimal or octal integer</td>
<td>Pointer to int</td>
</tr>
<tr>
<td>I</td>
<td>Decimal, hexadecimal or octal integer</td>
<td>Pointer to long</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>Pointer to unsigned int</td>
</tr>
<tr>
<td>U</td>
<td>Unsigned decimal integer</td>
<td>Pointer to unsigned long</td>
</tr>
<tr>
<td>e</td>
<td>Floating-point value consisting of an optional sign (+ or -), a series of one or more decimal digits possibly containing a decimal point, and an optional exponent (“e” or “E”) followed by an optionally signed integer value</td>
<td>Pointer to float</td>
</tr>
</tbody>
</table>

\(^1\) Since the input for a %x or %X format specifier is always interpreted as a hexadecimal number, the input should not include a leading 0x or 0X. (If 0x or 0X is included, the 0 is interpreted as a hexadecimal input value.)
<table>
<thead>
<tr>
<th>Character</th>
<th>Type of Input Expected</th>
<th>Type of Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Character. White-space characters that are ordinarily skipped are read when c is specified; to read the next non-white-space character, use %1s.</td>
<td>Pointer to char</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Pointer to character array large enough for input field plus a terminating null character ('\0'), which is automatically appended</td>
</tr>
<tr>
<td>n</td>
<td>No input read from stream or buffer</td>
<td>Pointer to int, into which is stored the number of characters successfully read from the stream or buffer up to that point in the current call to scanf</td>
</tr>
<tr>
<td>p</td>
<td>Value in the form xxxx:yyyy, where the digits x and y are uppercase hexadecimal digits</td>
<td>Pointer to far data item</td>
</tr>
</tbody>
</table>

To read strings not delimited by space characters, a set of characters in brackets ([ ]) can be substituted for the s (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing a terminating null character ('\0'), use the specification %nc, where n is a decimal integer. In this case, the c type character indicates that the argument is a pointer to a character array. The next n characters are read from the input stream into the specified location, and no null character ('\0') is appended. If n is not specified, the default value for it is 1.

The scanf function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons: the specified width has been reached; the next character cannot be converted as specified; the next character conflicts with a character in the control string that it is supposed to match; or the next character fails to appear (or does appear) in a given character set. When this occurs, the next input field is considered to begin at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first
character in subsequent read operations on stdin.

- **Return Value**

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

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

- **See Also**

  `fscanf`, `printf`, `sscanf`, `vfprintf`, `vprintf`, `vsscanf`

- **Examples**

  ```c
  #include <stdio.h>
  
  FILE *stream;
  int i;
  float fp;
  char c, s[81];

  main()
  {
    int result;

    printf("Enter an integer, a floating point number, \n    a character and a string\n>> ");
    result = scanf("%d %f %c %s", &i, &fp, &c, s);

    printf("\nThe number of fields input is %d\n", result);
    printf("The contents are: %d %f %c %s\n", i, fp, c, s);
  }
  
  Example 1 uses `scanf` to read various types of data from stdin.
  ```

  ```c
  #include <stdio.h>
  
  main()
  ```


```c
{  
    int numassigned, val;

    printf("Enter hexadecimal or octal #, or 00 to quit:\n");

    do
    {
        printf("# = ");

        /* Input octal or hex value: */
        numassigned = scanf("%i", &val);
        printf("Decimal # = %i\n", val);
    }
    while (val && numassigned);
    /* Loop ends if input value is 00 or */
    /* "scanf" is unable to assign field */
}
```

Sample output:

Enter hexadecimal or octal #, or 00 to quit:
# = 0xf
Decimal # = 15
# = 0100
Decimal # = 64
# = 00
Decimal # = 0

Example 2 uses `scanf` to read hexadecimal and octal values. It uses `printf` to convert these values to decimal and display them.
**Summary**

```
#include <dos.h>

void segread(segregs);
struct SREGS *segregs;    // Segment-register values
```

**Description**

The `segread` function fills the structure pointed to by `segregs` with the current contents of the segment registers. This function is intended to be used with the `intdosx` and `int86x` functions to retrieve segment register-values for later use.

**Return Value**

There is no return value.

**See Also**

`intdosx`, `int86x`, `FP_SEG`

**Example**

```
#include <dos.h>

struct SREGS segregs;
unsigned int cs, ds, es, ss;

main()
{
    segread(&segregs);           // * Read the values of *
    /* the segment registers */
    
    cs = segregs.cs;
    ds = segregs.ds;
    es = segregs.es;
    ss = segregs.ss;

    printf("cs = %x, ds = %x, es = %x, ss = %x\n", cs, ds, es, ss);
}
```
segread

This program uses segread to obtain the current values of the segment registers, then displays these values.
# include <stdio.h>

void setbuf(stream, buffer);

FILE *stream; Pointer to file structure
char *buffer; User-allocated buffer

Description

The setbuf function allows the user to control buffering for the specified stream. The argument stream must refer to an open file before it has been read or written. If the buffer argument is NULL, the stream is unbuffered. If not, the buffer must point to a character array of length BUFSIZ, where BUFSIZ is the buffer size as defined in stdio.h. The user-specified buffer is used for I/O buffering instead of the default system-allocated buffer for the given stream.

The stderr and stdaux streams are unbuffered by default but may be assigned buffers with setbuf.

Return Value

There is no return value.

See Also

fflush, fopen, fclose

Example

#include <stdio.h>

char buf[BUFSIZ];
FILE *stream1, *stream2;

main()
{
    stream1 = fopen("data1","r");
    stream2 = fopen("data2","w");

    /* "stream1" uses user-assigned buffer: */
setbuf

    setbuf(stream1,buf);
    /* "stream2" is unbuffered */
    setbuf(stream2,NULL);
    printf("Buffering of the streams has been set");
}

This program first opens files named DATA1 and DATA2. Then it uses setbuf to give DATA1 a user-assigned buffer and to change DATA2 so that it has no buffer.
Summary

#include <setjmp.h>

int setjmp(env);
jmp_buf env; Variable in which environment is stored

Description

The setjmp function saves a stack environment that can subsequently be restored using longjmp. Used together this way, setjmp and longjmp provide a way to execute a nonlocal goto and are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to setjmp causes the current stack environment to be saved in env. A subsequent call to longjmp restores the saved environment and returns control to the point just after the corresponding setjmp call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when longjmp was called. The values of register variables are unpredictable.

Return Value

The setjmp function returns 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 zero if the value argument of longjmp is 1. There is no error return.

See Also

longjmp

Warning

The values of register variables in the routine calling setjmp may not be restored to the proper values after a longjmp call is executed.
**setjmp**

- **Example**

```c
#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() {
    int error = 0;
    ...
    ...
    ...
    if (error != 0)
        longjmp(mark,-1);
    ...
    ...
}

recover() {
/* ensure that data files won't be corrupted by
** exiting the program. */
}
```
This program uses `setjmp` to save the stack environment and executes the function to simulate an error. It then uses `longjmp` to restore the stack environment and resume execution immediately after the `setjmp` call. Because `longjmp` and `setjmp` return different values, a conditional expression in the program allows the program to call the `recover` function to use additional error-recovery code.
setmode

- Summary

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

int setmode(handle, mode);
int handle;
int mode;
```

- Description

The `setmode` function sets the translation mode of the file given by `handle` to `mode`. The `mode` must be one of the following manifest constants:

<table>
<thead>
<tr>
<th>Manifest Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_TEXT</td>
<td>Set text (translated) mode. Carriage-return–line-feed combinations (CR-LF) are translated into a single line feed (LF) on input. Line-feed characters are translated into carriage-return–line-feed combinations on output.</td>
</tr>
<tr>
<td>Q_BINARY</td>
<td>Set binary (untranslated) mode. The above translations are suppressed.</td>
</tr>
</tbody>
</table>

The `setmode` function is typically used to modify the default translation mode of `stdin`, `stdout`, `stderr`, `stdaux`, and `stdprn`, but can be used on any file.

- Return Value

If successful, `setmode` returns the previous translation mode. A return value of -1 indicates an error, and `errno` is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid <code>mode</code> argument (neither Q_TEXT nor Q_BINARY)</td>
</tr>
</tbody>
</table>
See Also
creat, fopen, open

Example

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

int result;

main()
{
    /* Set "stdin" to have binary mode */
    /* (Initially "stdin" is in text mode): */
    result = setmode(fileno(stdin), O_BINARY);
    if ( result == -1 )
        perror("Cannot set mode");
    else
        printf(
            "'stdin' successfully changed to binary mode"
        );
}
```

This program uses `setmode` to change `stdin` from text mode to binary mode.
setvbuf

■ Summary

#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 (no buffer)
    _IOFBF (full buffering)
sizet size; Size of buffer

■ Description

The setvbuf function allows the user to control both buffering and buffer size for the specified stream. The stream must refer to an open file. The array that buf points to is used as the buffer, unless it is NULL, in which case the an automatically allocated buffer is used. The type specified by type is used; the type must be either _IONBF or _IOFBF. If type is _IOFBF, then size is used as the size of the buffer. If type is _IONBF, then the stream is unbuffered, and size and buf are ignored, as shown by the following:

<table>
<thead>
<tr>
<th>Type Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_IONBF</td>
<td>No buffer is used, regardless of buf or size.</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>Full buffering (unless buf is NULL); that is, use buf as the buffer and size as the size of the buffer.</td>
</tr>
</tbody>
</table>

The legal values for size are greater than 0 and less than the maximum integer size.

■ Return value

The return value for setvbuf is 0 if successful, and nonzero if an illegal type or buffer size is specified.
See Also

setbuf, fflush, fopen, fclose

Example

#include <stdio.h>

char buf[1024];
FILE *stream1, *stream2;

int result;

main()
{
  stream1 = fopen("data1","r");
  stream2 = fopen("data2","w");

  if (result = setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
    printf("Incorrect type or size of buffer1\n");
  else
    printf("'stream1' now has a buffer of 1024 bytes\n");

  if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
    printf("Incorrect type or size of buffer1\n");
  else
    printf("'stream2' now has no buffer\n");
}

This program opens two streams named stream1 and stream2. It then uses setvbuf to give stream1 a user-defined buffer of 1024 bytes and stream2 no buffer at all.
signal

- Summary

```c
#include <signal.h>

void *signal(sig, func([sig][, subcode]));
int sig; /* Signal value */
void *func(); /* Function to be executed */
int subcode; /* Optional error subcode */
```

- Description

The `signal` function allows a process to choose one of three ways to handle an interrupt signal from the operating system.

The `sig` argument must be one of the manifest constants listed below (defined in `signal.h`):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Corresponds to the DOS interrupt signal, INT 23H.</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Corresponds to floating-point exceptions that are not masked, such as overflow, division by zero, and invalid operation.</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>Abnormal termination, such as is caused by the <code>abort</code> function.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Invalid function image detected.</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Invalid storage access.</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Termination request sent to the program.</td>
</tr>
</tbody>
</table>

The `func` argument must be one of the manifest constants `SIG_DFL` or `SIG_IGN` (also defined in `signal.h`), or a function address. The action taken when the interrupt signal is received depends on the value of `func`, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIG_DFL</td>
<td>(Default) The calling process is terminated and control returns to the DOS command level. All files opened by the process are closed, but buffers are not flushed.</td>
</tr>
</tbody>
</table>
SIG_IGN  The interrupt signal is ignored. This value should never be given for SIGFPE, since the floating-point state of the process is left undefined.

Function address  For SIGINT signals, the function pointed to by func is passed the single argument SIGINT and executed. If the function returns, the calling process resumes execution immediately following the point where it received the interrupt signal. Before the specified function is executed, the value of func is set to SIG_DFL; the next interrupt signal is treated as described above for SIG_DFL, unless an intervening call to signal specifies otherwise. This allows the user to reset signals in the called function if desired.

For SIGFPE, the function pointed to by func is passed two arguments, SIGFPE and an integer error subcode, FPE_xxx, then executed. (See the include file float.h for definitions of the FPE_xxx subcodes.) The second value is not part of the ANSI standard; it is a DOS extension. The value of func is not reset upon receiving the signal; to recover from floating-point exceptions, use setjmp in conjunction with longjmp. (See the example under _fpreset in this Reference.) If the function returns, the calling process resumes execution with the floating-point state of the process left in an undefined state.

Return Value

The signal function returns the previous value of func. A return value of SIG_ERR indicates an error, and errno is set to EINVAL, indicating an invalid sig value.
signal

• See Also

abort, exit, _exit, _fpreset, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

---

Note

Signal settings are not preserved in child processes created by calls to exec or spawn routines. The signal settings are reset to the default in the child process.

---

• Example

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

int handler();

main()
{
    /* Set so interrupt calls "handler": */
    if (signal(SIGINT, handler) == (int(*)(void))-1)
    {
        fprintf(stderr,"Couldn't set SIGINT");
        abort();
    }
    for (;;) printf("Hit control C:\n");
}

/* Function called at OS interrupt */
int handler()
{
    char ch;

    /* Disallow ctrl-c during handler: */
    signal(SIGINT, SIG_IGN);

    printf("Terminate processing? ");
    ch = getch();
    if ((ch == 'y') || (ch == 'Y')) exit(0);
```
This program uses `signal` to set up the `handler` function as the routine that is called to execute an operating-system interrupt. When the user presses CTRL-C, `handler` is called to handle the interrupt.
sin – sinh

■ Summary

#include <math.h>

double sin(x);   // Calculate sine of x
double sinh(x);  // Calculate hyperbolic sine of x
double x;       // Radians

■ Description

The sin and sinh functions return the sine and hyperbolic sine of x (measured in radians), respectively.

■ Return Value

The sin function returns the sine of x. If x is large, a partial loss of significance in the result may occur. 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 (positive or negative, depending on the value of x).

Error handling can be modified by using the matherr routine.

■ See Also

acos, asin, atan, atan2, cos, cosh, tan, tanh

■ Example

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

main()
{
  double pi = 3.1415926535, x,y;
}
\[
x = \pi/2;
\]
\[
y = \sin(x);
\text{printf}("The sin(\%f) = \%f\n", x, y);
\]
\[
y = \sinh(x);
\text{printf}("The sinh(\%f) = \%f\n", x, y);
\]

This program displays the sine and hyperbolic sine of π/2.
sopen

- Summary

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <share.h>
#include <io.h>
```

int sopen(pathname, oflag, shflag[, pmode]);

```c
char *pathname;          // File path name
int oflag;              // Type of operations allowed
int shflag;             // Type of sharing allowed
int pmode;              // Permission setting
```

- Description

The `sopen` function opens the file specified by `pathname` and prepares the file for subsequent shared reading or writing, as defined by `oflag` and `shflag`. The integer expression `oflag` is formed by combining one or more of the following manifest constants, defined in `<fcntl.h>`. When more than one manifest constant is given, the constants are joined with the OR operator (|).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>O_APPEND</code></td>
<td>Reposition the file pointer to the end of the file before every write operation.</td>
</tr>
<tr>
<td><code>O_CREAT</code></td>
<td>Create and open a new file; this has no effect if the file specified by <code>pathname</code> exists.</td>
</tr>
<tr>
<td><code>O_EXCL</code></td>
<td>Return an error value if the file specified by <code>pathname</code> exists; applies only when used with <code>O_CREAT</code>.</td>
</tr>
<tr>
<td><code>O_RDONLY</code></td>
<td>Open file for reading only; if this flag is given, neither <code>O_RDWR</code> nor <code>O_WRONLY</code> can be given.</td>
</tr>
<tr>
<td><code>O_RDWR</code></td>
<td>Open file for both reading and writing; if this flag is given, neither <code>O_RDONLY</code> nor <code>O_WRONLY</code> can be given.</td>
</tr>
<tr>
<td><code>O_TRUNC</code></td>
<td>Open and truncate an existing file to 0 bytes; the file must have write permission; the contents of the file are destroyed.</td>
</tr>
</tbody>
</table>
**sopen**

O_ WRONLY

Open file for writing only; if this flag is given, neither O_RDONLY nor O_RDWR can be given.

O_BINARY

Open file in binary (untranslated) mode. (See fopen for a description of binary mode.)

O_TEXT

Open file in text (translated) mode. (See fopen for a description of text mode.)

---

**Note**

O_TRUNC destroys the entire contents of an existing file. Use with care.

---

The argument shflag is a constant expression consisting of one of the following manifest constants, defined in share.h. If SHARE.COM (or SHARE.EXE for some versions of DOS) is not installed, DOS ignores the sharing mode. (See your DOS documentation for detailed information about sharing modes.)

<table>
<thead>
<tr>
<th>shflag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH_COMPAT</td>
<td>Set compatibility mode</td>
</tr>
<tr>
<td>SH_DENYRW</td>
<td>Deny read and write access to file</td>
</tr>
<tr>
<td>SH_DENYWR</td>
<td>Deny write access to file</td>
</tr>
<tr>
<td>SH_DENYRD</td>
<td>Deny read access to file</td>
</tr>
<tr>
<td>SH_DENYNO</td>
<td>Permit read and write access</td>
</tr>
</tbody>
</table>

The pmode argument is required only when O_CREAT is specified. If the file does not exist, pmode specifies the file's permission settings, which are set when the new file is closed for the first time. Otherwise, the pmode argument is ignored. The pmode argument is an integer expression 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 OR operator (|). The meaning of the pmode argument is as follows:
### Value

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IWRITE</td>
<td>Writing permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>Reading permitted</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IWRITE</td>
</tr>
</tbody>
</table>

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.

---

### Important

Under DOS versions 3.0, 3.1 and 3.2 with SHARE installed, a bug occurs when opening a new file with omode set to O_CREAT | O_RDONLY or O_CREAT | WRONLY, pmode set to S_IREAD, and shflag set to SH_COMPAT. In this case, the operating system will prematurely close the file during system calls made within sopen, or chmod will generate a sharing violation (INT 24H).

To get around the problem, open the file with pmode set to S_IWRITE. After closing the file, call chmod and change the mode back to S_IREAD. Another work-around is to open the file with pmode set to S_IREAD, oflag set to O_CREAT | O_RDWR, and shflag set to SH_COMPAT.

---

The sopen function applies the current file-permission mask to pmode before setting the permissions (see umask).

#### Return Value

The sopen function returns a file handle for the opened file. A return value of -1 indicates an error, and errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Given path name is a directory; or the file is read only but an open for writing was attempted; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; DOS.</td>
</tr>
</tbody>
</table>
sopen

version 3.0 or later only).

EEXIST
The **O_CREAT** and **O_EXCL** flags are specified, but the named file already exists.

EMFILE
No more file handles available (too many open files).

ENOENT
File or path name not found.

- See Also

close, creat, fopen, open, umask

---

**Note**

The **sopen** function should be used only under DOS Version 3.0 or later. Under earlier versions of DOS, the **shflag** argument is ignored.

File sharing modes will not work correctly for buffered files, so do not use **fdopen** to associate a file opened for sharing (or locking) with a stream.

---

**Example**

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

extern unsigned char _osmajor;

int fh;

main()
{
    /* Open for file sharing: */
    if (_osmajor >= 3)
        fh = sopen("sopen.c", O_RDWR | O_BINARY, SH_DENYRW);

    /* Just a regular open */
    else
        fh = open("sopen.c", O_RDWR | O_BINARY);
```

441
This program first checks the version of DOS. If the version is 3.0 or later, it uses `sopen` to open a file named `SOPEN.C` for sharing.
# Summary

```c
#include <stdio.h>
#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);

int modeflag;
char *pathname;
char *arg0, *arg1..., *argn;
char *argv[ ];
char *envp[ ];
```

- **Execution mode for parent process**
- **Path name of file to be executed**
- **List of pointers to arguments**
- **Array of pointers to arguments**
- **Array of pointers to environment settings**

## Description

The `spawn` functions create and execute a new child process. Enough memory must be available for loading and executing the child process. The `modeflag` argument determines the action taken by the parent process before and during the `spawn`. The following values for `modeflag` are defined in `process.h`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>P - WAIT</td>
<td>Suspend parent process until execution of child process is complete (synchronous <code>spawn</code>)</td>
</tr>
<tr>
<td>P - OVERLAY</td>
<td>Overlay parent process with child, destroying the parent (same effect as <code>exec</code> calls)</td>
</tr>
</tbody>
</table>
The *pathname* argument specifies the file to be executed 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 file name. If *pathname* does not have a file-name extension or end with a period (.), search for the file; if unsuccessful, the extension .EXE is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the *spawn* calls search for *pathname* with no extension. The *spawnlp*, *spawnlpe*, *spawnvp*, and *spawnvpe* routines search for *pathname* (using the same procedures) in the directories specified by the **PATH** environment variable.

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the *spawn* call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes. The terminating null character ("\0") for each string is not included in the count, but space characters (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* must be passed to the child process (which sees it as *argv[0]*). Usually, this argument is a copy of the *pathname* argument. (A different value will not produce an error.) Under versions of DOS earlier than 3.0, the passed value of *arg0* is not available for use in the child process. However, under DOS 3.0 and later, the *pathname* is available as *arg0*.

The *spawnl*, *spawnle*, *spawnlp*, and *spawnlpe* calls are typically used in cases where the number of arguments is known in advance. The *arg0* argument is usually a pointer to *pathname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a **NULL** pointer to mark the end of the argument list.

The *spawnv*, *spawnve*, *spawnvp*, and *spawnvpe* calls are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. The argument *argv[0]* is usually a pointer to the *pathname* and *argv[1]* through *argv[n]* are pointers to the character strings forming the new argument list. The argument *argv[n+1]* must be a **NULL** pointer to mark the end of the argument list.
Files that are open when a spawn call is made remain open in the child process. In the spawnl, spawnlp, spawnv, and spawnvp calls, the child process inherits the environment of the parent. The spawnle, spawnlp, spawnve, and spawnvpe calls allow the user to alter the environment for the child process by passing a list of environment settings through the envp argument. The argument envp is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

\[
\text{NAME}=\text{value}
\]

where NAME is the name of an environment variable and value is the string value to which that variable is set. (Note that value is not enclosed in double quotes.) The final element of the envp array should be NULL. When envp itself is NULL, the child process inherits the environment settings of the parent process.

The spawn functions pass the child process all information about open files, including the translation mode, through the ;C_FILE_INFO entry in the environment that is passed. The C start-up code normally processes this entry and then deletes it from the environment. However, if a spawn function spawns a non-C process (such as COMMAND.COM), this entry will remain in the environment. In this case, since the environment information is passed in binary form, printing the environment will show graphics characters in the definition string for this entry. It has no other effect on normal operations.

- **Return Value**

The return value from a synchronous spawn (P_WAIT specified for modeflag) is the exit status of the child process.

The return value from an asynchronous spawn (P_NOWAIT specified for modeflag) is the process ID. To obtain the exit code for the spawned process, you must call the wait or cwait function and specify the process ID.

The exit status is 0 if the process terminated normally. The exit status can be set to a nonzero value if the child process specifically calls the exit routine with a nonzero argument. If the child process did not set a positive exit status, the positive exit status indicates an abnormal exit with an abort or an interrupt.
spawnl – spawnvpe

A return value of -1 indicates an error (the child process is not started). In this case, \texttt{errno} is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid \textit{modeflag} argument.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>The specified file is not executable or has an invalid executable-file format.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory is available to execute the child process.</td>
</tr>
</tbody>
</table>

\section*{Note}

Signal settings are not preserved in child processes created by calls to \texttt{spawn} routines. The signal settings are reset to the default in the child process.

\section*{See Also}

\texttt{abort, execl, execle, execlp, execle, execv, execve, execvp, execvpe, exit, \_exit, onexit, system}
Example

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

char *my_env[] = {
    "THIS=environment will be",
    "PASSED=to child.exe by the",
    "SPAWNLE=and",
    "SPAWNLPE=and",
    "SPAWNVE=and",
    "SPAWNVPE=functions",
    NULL
};

main(argc, argv)
int argc;
char *argv[];
{
    char *args[4];
    int result;

    args[0] = "child"; /* Set up parameters to send */
    args[1] = "spawn??";
    args[2] = "two";
    args[3] = NULL;

    switch (argv[1][0])  /* Based on first letter of argument */
    {
        case '1':
            spawnl (P_WAIT, "child.exe","child ","spawnl",
                    "two",NULL);
            break;
        case '2':
            spawnle (P_WAIT, "child.exe","child","spawnle",
                    "two",NULL,my_env);
            break;
        case '3':
            spawnlp (P_WAIT, "child.exe","child","spawnlp",
                    "two",NULL);
            break;
        case '4':
            spawnlpe(P_WAIT, "child.exe","child","spawnlpe",
                    "two",NULL,my_env);
            break;
        case '5':
            spawnv (P_OVERLAY, "child.exe",args);
            break;
        case '6':
            break;
    }
}
```
spawnl – spawnvpe

    spawnve (P_OVERLAY, "child.exe", args, my_env);
    break;
    case '7':
        spawnvp (P_OVERLAY, "child.exe", args);
    break;
    case '8':
        spawnvpe (P_OVERLAY, "child.exe", args, my_env);
    break;
    default:
        printf("Enter a number from 1 to 8 as a command line parameter.");
        exit();
    }

    printf("\n\nReturned from SPAWN!\n");
}

This program accepts a number in the range 1 through 8 from the command line. Based on the number it receives, it executes one of the eight different procedures that spawn the process named child. For some of these procedures, the CHILD.EXE file must be in the same directory; for others, it must only be in the same path.
Summary

#include <stdio.h>

int sprintf(buffer, format-string[, argument...]);
char *buffer; // Storage location for output
const char *format-string; // Format-control string

Description

The sprintf function formats and stores a series of characters and values in buffer. Each argument (if any) is converted and output according to the corresponding format specification in the format-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 reference page for a description of the format-string and arguments. A null character is appended to the end of the characters written but is not counted in the return value.

Return Value

The sprintf function returns the number of characters stored in buffer, not counting the terminating null.

See Also

fprintf, printf, sscanf

Example

#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "computer";
char c;

main()
{}
c = 'l';
i = 35;
printf

fp = 1.7320508;

/* Format and print various data: */
j = sprintf(buffer, "%s\n", s);
j += sprintf(buffer+j, "%c\n", c);
j += sprintf(buffer+j, "%d\n", i);
j += sprintf(buffer+j, "%f\n", fp);

printf("string: \n%s\ncharacter count = %d\n", buffer, j);
}

This program uses `sprintf` to format various data and place them in the string named `buffer`.
# Summary

#include <math.h>

double sqrt(x);
double x; Non-negative floating-point value

## Description

The sqrt function calculates the square root of x.

## Return Value

The sqrt function returns the square root result. If x is negative, the function prints a DOMAIN error message to stderr, sets errno to EDOM, and returns 0.

Error handling can be modified by using the matherr routine.

## See Also

exp, log, matherr, pow

## Example

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

main ()
{
    double x, y, z;
    x = 1.0;
    y = 3.0;

    if ((z = sqrt(x+y)) == 0.0) /* Return of 0 means arg<0 */
    {
        if ((x+y) < 0.0) /* z is 2 */
            perror("sqrt of a negative number");
    }
    else
        printf("The square root of %f = %f\n", x+y, z);
}
This program uses \texttt{sqrt} to display the square root of 4.
srand

■ Summary

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

void srand(seed);
unsigned seed;  // Seed for random-number generation
```

■ Description

The `srand` function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the `seed` argument. Any other value for `seed` sets the generator to a random starting point.

The `rand` function is used to retrieve the pseudorandom numbers generated. If `rand` is called before any calls are made to `srand`, the same sequence is generated that would have been had `srand` been called with a `seed` value of 1.

■ Return Value

There is no return value.

■ See Also

rand

■ Example

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

main()
{
    int x, ranvals[20];

    srand(17);
    /* Initialize array and output values: */
    for (x = 0; x < 20; ranvals[x++] = rand())
        printf("Iteration %d, ranvals[%d] =%d\n",x+1,x,ranvals[x]);
}
```

First, this program calls `srand` with a value other than 1 to randomize a random-value sequence. Then it initializes an array named `ranvalues` with 20 random values.
sscanf

- Summary

```c
#include <stdio.h>

int sscanf(const char *buffer, const char *format-string, argument ...);
```

- Description

The `sscanf` function reads data from `buffer` 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 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`.

- Return Value

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

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

- See Also

`fscanf, scanf, sprintf`
Example

#include <stdio.h>

char *tokenstring = "15 12 14...";
int i;
float fp;
char s[81];
char c;

main()
{
    /* Input various data from tokenstring: */
    sscanf(tokenstring, "%s", s);
    sscanf(tokenstring, "%c", &c);
    sscanf(tokenstring, "%d", &i);
    sscanf(tokenstring, "%f", &fp);

    /* Output the data read */
    printf("string =%s\n", s);  /* s is 15 */
    printf("character =%c\n", c); /* c is 1 */
    printf("integer =%d\n", i); /* i is 15 */

    /* fp is 15.000000 */
    printf("floating point number =%f\n", fp);
}

This program uses scanf to read various data from a string named tokenstring, then displays it.
stackavail

- Summary

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

unsigned int stackavail();
```

- Description

The **stackavail** function returns the approximate size in bytes of the stack space available for dynamic memory allocation with **alloca**.

- Return Value

The **stackavail** function returns the size in bytes as an unsigned integer value.

- See Also

alloca, freect, memavl

- Example

```
#include <malloc.h>
main()
{
    char *ptr;

    printf("Stack memory available before alloc = %u\n", stackavail());
    ptr = alloca(1000*sizeof(char));
    printf("Stack memory available after alloc = %u\n", stackavail());
}
```

Sample output:

```
Stack memory available before alloc = 1682
Stack memory available after alloc = 678
```
This program uses `stackavail` to determine the amount of free space available on the stack. It then allocates memory from the stack and calls `stackavail` again to display the new amount of available free space.
stat

- **Summary**

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

int stat(pathname, buffer);
char *pathname;               // Path name of existing file
struct stat *buffer;         // Pointer to structure to receive results
```

- **Description**

The `stat` function obtains information about the file or directory specified by `pathname` and stores it in the structure pointed to by `buffer`. The `stat` structure, defined in `sys/stat.h`, contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>st_mode</code></td>
<td>Bit mask for file-mode information. <code>S_IFDIR</code> bit set if <code>pathname</code> specifies a directory; <code>S_IFREG</code> bit set if <code>pathname</code> specifies an ordinary file. User read/write bits set according to the file's permission mode; user execute bits set using the file-name extension.</td>
</tr>
<tr>
<td><code>st_dev</code></td>
<td>Drive number of the disk containing the file. (same as <code>st_dev</code>).</td>
</tr>
<tr>
<td><code>st_rdev</code></td>
<td>Drive number of the disk containing the file (same as <code>st_dev</code>).</td>
</tr>
<tr>
<td><code>st_nlink</code></td>
<td>Always 1.</td>
</tr>
<tr>
<td><code>st_size</code></td>
<td>Size of the file in bytes.</td>
</tr>
<tr>
<td><code>st_atime</code></td>
<td>Time of last modification of file. (same as <code>st_mtime</code> and <code>st_ctime</code>).</td>
</tr>
<tr>
<td><code>st_mtime</code></td>
<td>Time of last modification of file (same as <code>st_atime</code> and <code>st_ctime</code>).</td>
</tr>
<tr>
<td><code>st_ctime</code></td>
<td>Time of last modification of file (same as <code>st_atime</code> and <code>st_mtime</code>).</td>
</tr>
</tbody>
</table>

There are three additional fields in the `stat` structure type that do not contain meaningful values under DOS.
Return Value

The stat function returns the value 0 if the file-status information is obtained. A return value of -1 indicates an error, and errno is set to ENOENT, indicating that the file name or path name could not be found.

See Also

access, fstat

Note

If the given pathname refers to a device, the size and time fields in the stat structure are not meaningful.

Example

```c
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";

main()
{
    /* Get data associated with "data": */
    result = stat("data", &buf);

    /* Check if statistics are valid: */
    if (result != 0)
        perror("Problem getting information ");
    else
    {
        printf("File size : %ld\n", buf.st_size);
        printf("Drive number : %d\n", buf.st_dev);
        printf("Time modified : %s", ctime(&buf.st_atime));
    }
}
```
This program uses `stat` to report the size, drive number, and last modification time for the file named DATA.
Summary

```c
#include <float.h>

unsigned int _status87();
```

Get floating-point status word

Description

The `_status87` function gets the floating-point status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

Return Value

The bits in the value returned indicate the floating-point status. See the `float.h` include file for a complete definition of the bits returned by `_status87`.

---

Note

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

---

See Also

`_clear87, _control87`
Example

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

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

main()
{
    printf("Status = %.4x - clear\n", _status87());

    /* Store into y is inexact & underflows: */
    y = a;
    printf("Status = %.4x - inexact, underflow\n", _status87());

    /* y is denormal: */
    b = y;
    printf("Status = %.4x - inexact underflow, denormal\n",
           _status87());

    /* Clear user 8087: */
    _clear87();
}
```

This program creates various floating-point errors and then uses _status87 to display messages indicating these problems.
<table>
<thead>
<tr>
<th><strong>Summary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code># include &lt;string.h&gt;</code></td>
</tr>
<tr>
<td><code>char *strcat(string1, string2);</code></td>
</tr>
<tr>
<td><code>char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
<tr>
<td><code>char *strchr(string, c);</code></td>
</tr>
<tr>
<td><code>const char *string;</code></td>
</tr>
<tr>
<td><code>int c;</code></td>
</tr>
<tr>
<td><code>int strcmp(string1, string2);</code></td>
</tr>
<tr>
<td><code>const char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
<tr>
<td><code>int strcmpi(string1, string2);</code></td>
</tr>
<tr>
<td><code>const char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
<tr>
<td><code>char strcpy(string1, string2);</code></td>
</tr>
<tr>
<td><code>char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
<tr>
<td><code>size_t strcspn(string1, string2);</code></td>
</tr>
<tr>
<td><code>const char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
<tr>
<td><code>char *strdup(string);</code></td>
</tr>
<tr>
<td><code>const char *string;</code></td>
</tr>
<tr>
<td><code>int strcmp(string1, string2);</code></td>
</tr>
<tr>
<td><code>const char *string1;</code></td>
</tr>
<tr>
<td><code>const char *string2;</code></td>
</tr>
</tbody>
</table>
strcat – strdup

- Description

The `strcat`, `strchr`, `strcmp`, `strcmpl`, `strcpyl`, `strcspn`, `strdup`, and `stricmp` functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character (`\0`) marking the end of the string. No overflow checking is performed when strings are copied or appended.

The `strcat` function appends `string2` to `string1`, terminates the resulting string with a null character, and returns a pointer to the concatenated string (`string1`).

The `strchr` function returns a pointer to the first occurrence of `c` in `string`. The character `c` may be the null character (`\0`); the terminating null character of `string` is included in the search. The function returns `NULL` if the character is not found.

The `strcmp` function compares `string1` and `string2` lexicographically and returns a value indicating their relationship, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> identical to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

The `strcmpl` 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 `strcpyl` function copies `string2`, including the terminating null character, to the location specified by `string1`, and returns `string1`.

The `strcspn` function returns the index of the first character in `string1` that belongs to the set of characters specified by `string2`. This value is equivalent to the length of the initial substring of `string1` that consists entirely of characters not in `string2`. Terminating null characters are not considered in the search. If `string1` begins with a character from `string2`, `strcspn` returns 0.
The `strdup` function allocates storage space (with a call to `malloc`) for a copy of `string` and returns a pointer to the storage space containing the copied string. The function returns `NULL` if storage could not be allocated.

- **Return Value**

The return values for these functions are described above.

- **See Also**

`strncat`, `strncmp`, `strncpy`, `strnicmp`, `strrchr`, `strspn`

- **Example**

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

char string[100] = "XYZabc This is a string!";
char template[100] = "XYZabc This is A STRING!";
char *newstring;
char *result;
int numresult;

main()
{
    /* Construct "computer program using "strcpy" and "strcat" */
    strcpy(string, "computer");
    result = strcat(string, " program");
    printf("Result = \%s\n", result);

    /* Find the first occurrence of 'a': */
    result = strchr(string, 'a');
    printf("String after an \"a\" is \%s\n", result);

    /* Compare one string against another */
    /* and report whether less than, greater than */
    /* or equal to: */
    numresult = strcmp(string, template);
    printf( "\"\%s\" is \%s \"\%s\"\n", string, numresult ?
            ( numresult > 0 ? "greater than": "less than") ;
            "equal to", template );

    /* Compare string with regard to case */
```
**strcat – strdup**

```c
numresult = strcmpi("hello", "HELLO");
printf(""%s" is %s "%s"
", "hello", numresult ?
    (numresult > 0 ? "greater than" : "less than" ) :
    "equal to", "HELLO" );

/* Make a copy of a string */
printf(""%s" "%s"
", template, string);
result = strcpy(template, string);
printf(""%s" "%s"
", template, string);

strcpy(string, "xyzabc"); /* Search for a's, b's, or c's */
numresult = strcspn(string, "abc");
printf("The location of the first a, b, or c is %d
", numresult );

/* Make newstring point to a duplicate of string: */
newstring = strdup(string);
printf("The new string is %s
", newstring);
}
```

This program demonstrates the uses of the `strcat`, `strch`, `strcmp`, `strcmpi`, `strcpy`, `strcspn`, and `strdup` functions.
### Summary

```c
#include <string.h>
char *strerror(string);
```

Required only for function declarations

- `string`: User-supplied message
- `errno`: Error number
- `sys_nerr`: Number of system messages
- `sys_errlist`: Array of error messages

### Description

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; this string is terminated by the new-line character (`'
'`).

If `string` is not equal to `NULL`, then `strerror` returns a pointer to a string containing, in order, your string message, a colon, a space, the system error message for the last library call producing an error, and a new-line character. Your `string` message can be a maximum of 94 bytes long.

Unlike `perror`, `strerror` alone does not print any messages. To print the message returned by `strerror` to `stderr`, your program will need a `printf` statement, as shown in the following lines:

```c
if ((access("datafile", 2)) == -1)
    printf(strerror(NULL));
```

The actual error number is stored in the variable `errno`, which should be declared at the external level. The system error messages are accessed through the variable `sys_errlist`, which is an array of messages ordered by error number. The `strerror` function accesses the appropriate error message by using the `errno` value as an index to `sys_errlist`. The value of the variable `sys_nerr` is defined as the maximum number of elements in the `sys_errlist` array.

To produce accurate results, `strerror` should be called immediately after a library routine returns with an error. Otherwise, the `errno` value may be overwritten by subsequent calls.
**strerror**

- **Return Value**
  The `strerror` function returns no value.

- **See Also**
  `clearerr`, `ferror`, `perror`

---

**Note**

Under DOS, some of the `errno` values listed in `errno.h` are not used. See Appendix A, "Error Messages," for a list of `errno` values used on DOS, and the corresponding error messages. The `strerror` function prints an empty string for any `errno` value not used under DOS.

---

- **Example**

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

main()
{
    /* Since "xxxx" does not exist, */
    /* both open attempts will fail */
    int fh1, fh2;
    fh1 = open("xxxx", O_RDONLY);
    if (fh1 == -1)
        printf(strerror("Open failed on input file "));

    fh2 = open("xxxx", O_WRONLY|O_TRUNC, S_IWRITE|S_IWRITE);
    if (fh2 == -1)
        printf(strerror("Open failed on output file "));
}
```

This program tries to open files for input and output. If an error occurs, the program uses `strerror` to tag an error message onto the front of the standard error message and then displays the entire error message.
- **Summary**

```c
#include <string.h>
size_t strlen(string);
char *string;
```

- **Description**

The `strlen` function returns the length in bytes of `string`, not including the terminating null character (`'\0'`).

- **Return Value**

The `strlen` function returns the `string` length. There is no error return.

- **Example**

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

char *string = "some space";
size_t result;

main()
{
    result = strlen(string); /* result is 10 */
    printf("The size of the string is %d", result);
}
```

This program uses `strlen` to determine the length of the string named `string`. 
strlwr

- **Summary**

```c
#include <string.h>  // Required only for function declarations

char *strlwr(string);
```

- **Description**

The `strlwr` function converts any uppercase letters in the given null-terminated `string` to lowercase. Other characters are not affected.

- **Return Value**

The `strlwr` function returns a pointer to the converted `string`. There is no error return.

- **See Also**

`strupr`

- **Example**

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

char string[100] = "This Was a Mixed-Case String", *copy;

main()
{
    copy = strlwr(strdup(string));
    printf("The result string is: %s", copy);
}
```

This program duplicates a string named `string`, then uses `strlwr` to convert all uppercase letters in the copy to lowercase.
Summary

`# include <string.h>` Required only for function declarations

```c
char *strncat(string1, string2, n);
```
Append `n` characters of `string2` to `string1`

```c
char *string1;
```
Destination string

```c
const char *string2;
```
Source string

```c
size_t n;
```
Number of characters appended

```c
int strncmp(string1, string2, n);
```
Compare first `n` characters of strings

```c
const char *string1;
```
Source string

```c
const char *string2;
```
Source string

```c
size_t n;
```
Number of characters compared

```c
int strnicmp(string1, string2, n);
```
Compare first `n` characters of strings

```c
const char *string1;
```
Source string

```c
const char *string2;
```
Source string

```c
size_t n;
```
Number of characters compared

```c
char *strncpy(string1, string2, n);
```
Copy `n` characters of `string2` to `string1`

```c
char *string1;
```
Destination string

```c
const char *string2;
```
Source string

```c
size_t n;
```
Number of characters copied

```c
char *strnset(string, c, n);
```
Initialize first `n` characters of `string`

```c
char *string;
```
String to be initialized

```c
int c;
```
Character setting

```c
size_t n;
```
Number of characters set

Description

The `strncat`, `strncmp`, `strnicmp`, `strncpy`, and `strnset` functions operate on, at most, the first `n` characters of null-terminated strings.

The `strncat` function appends, at most, the first `n` characters of `string2` to `string1`, terminates the resulting string with a null character (`\0`), and returns a pointer to the concatenated string (`string1`). If `n` is greater than the length of `string2`, the length of `string2` is used in place of `n`. 
The **strncmp** function compares, at most, the first *n* characters of *string1* and *string2* lexicographically and returns a value indicating the relationship between the substrings, as listed below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><em>substring1</em> less than <em>substring2</em></td>
</tr>
<tr>
<td>0</td>
<td><em>substring1</em> equivalent to <em>substring2</em></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><em>substring1</em> greater than <em>substring2</em></td>
</tr>
</tbody>
</table>

The **strnicmp** function is a case-insensitive version of **strncmp**: **strnicmp** converts all alphabetic characters in the two strings *string1* and *string2* to lowercase before comparing them, so that all uppercase 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 appended automatically to the copied string. If *n* is greater than the length of *string2*, the *string1* result is padded with null characters ("\0") up to length *n*.

---

**Note**

The behavior of **strncpy** is undefined if the address ranges of *string1* and *string2* overlap.

---

The **strnset** function sets, at most, the first *n* characters of *string* to the character *c* and returns a pointer to the altered *string*. If *n* is greater than the length of *string*, the length of *string* is used in place of *n*.

**See Also**

**strcat, strcmp, strcpy, strset**
Example

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

char string[100] = "XYZabc This is a string!", *result;
char copy[100] = "This is a different string";
char *result;
char suffix[100] = " this is even more string..";
int numresult;

main()
{
    /* Combine strings with no more than */
    /* 100 characters of suffix: */
    printf("String before = %s\n", string);
    result = strncat(string, suffix, 100);
    printf("String after = %s\n", string);

    /* Determine ordering of two strings */
    /* but only consider first 7 chars: */
    strcpy(string, "programming");
    numresult = strncmp(string, "program", 7);
    printf("%s is %s "%s\n", string, numresult ? (numresult > 0 ?
        "greater than" : "less than") : "equal to", "program");

    /* Copy at most 99 chars of "string" */
    printf("%s" "%s"\n", copy, string);
    result = strncpy(copy, string, 99);
    copy[99] = '\0'; /* Null terminate the result */
    printf("%s" "%s"\n", copy, string);

    /* Set not more than 4 characters of a */
    /* string to be x's: */
    result = strncat("computer", 'x', 4);
    printf( "%s\n", result ); /* Result is now "xxxxuter". */
}
```

This program demonstrates the uses of the `strncat`, `strncmp`, `strnicmp`, and `strnset` functions.
strpbrk

- **Summary**

```c
#include <string.h>

char *strpbrk(const char *string1, const char *string2);
const char *string1;  // Source string
const char *string2;  // Character set
```

- **Description**

The `strpbrk` function finds the first occurrence in `string1` of any character from `string2`. The terminating null character ("\0") is not included in the search.

- **Return Value**

The `strpbrk` function returns a pointer to the first occurrence of any character from `string2` in `string1`. A NULL pointer indicates that `string1` and `string2` have no characters in common.

- **See Also**

`strchr`, `strrchr`

- **Example**

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

char string[100] = "Find an 'a' or 'b' in this string", *result;

main()
{
    /* Return pointer to first '
      * 'a' or 'b' in "string" */
    result = strpbrk(string, "ab");
    printf("The remainder of the string
          starting at the first\n");
    printf("first 'a' or 'b' is: %s", result);
}
```
This program uses \texttt{strpbrk} to find the first occurrence of \texttt{a} or \texttt{b} in the string named \texttt{string}. 
**strrchr**

- **Summary**

```c
#include <string.h>

char *strrchr(string, c);
const char *string;
int c;
```

- **Description**

The `strrchr` function finds the last occurrence of the character `c` in `string`. The `string`'s terminating null character (`'\0'`) is included in the search. (Use `strchr` to find the first occurrence of `c` in `string`.)

- **Return Value**

The `strrchr` function returns a pointer to the last occurrence of `c` in `string`. A NULL pointer is returned if the given character is not found.

- **See Also**

`strchr, strpbrk`

- **Example**

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

char string[100] = "Find the last 'a' in this string", *result;

main()
{
    /* Return a pointer to the last: 'a' */
    result = strrchr(string, 'a');
    printf("The remainder of the string starting at the first\n");
    printf("'a' is: %s", result);
}
```
This program uses `strrchr` to find the last occurrence of a in the string named `string`. 
**strrev**

- **Summary**

```c
#include <string.h>

char *strrev(string);
```

Required only for function declarations

- **Description**

The `strrev` function reverses the order of the characters in the given `string`. The terminating null character (`\0`) remains in place.

- **Return Value**

The `strrev` function returns a pointer to the altered `string`. There is no error return.

- **See Also**

`strcpy`, `strset`

- **Example**

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

char string[100];
int result;

main()
{
    printf("Input a string and I will tell you if it is a palindrome: ");
    gets(string);
    /* Reverse string and compare: */
    result = strcmp(string, strrev(strdup(string)));

    if (result == 0)
        printf("The string "%s" is a palindrome\n\n", string);
    else
        printf("The string "%s" is not a palindrome\n\n", string);
}```
This program checks an input string to see whether it is a palindrome: that is, whether it reads the same forward and backward. The program checks this by comparing a string named string with a copy of string that has been reversed using strrev.
strset

- **Summary**

```
#include <string.h>  

char *strset(string, c);  
char *string;  
int c;  
```

- **Description**

The `strset` function sets all characters of the given `string`, except the terminating null character (`\0`), to `c`.

- **Return Value**

The `strset` function returns a pointer to the altered `string`. There is no error return.

- **See Also**

`strnset`

- **Example**

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

cchar string[100] = "Fill the string with something", *result;

main()
{
    printf("The string before 'strset' is used: \"%s\"\n", string);
    result = strset(string, ' ');
    printf("The string after 'strset' was used: \"%s\"\n", string);
}
```

This program uses `strset` to fill the string named `string` with blanks.
Summary

#include <string.h>

  size_t strspn(string1, string2);
const char *string1;
const char *string2;

Description

The strspn function returns the index of the first character in string1 that does not belong to the set of characters specified by string2. This value is equivalent to the length of the initial substring of string1 that consists entirely of characters from string2. The null character ("\0") terminating string2 is not considered in the matching process. If string1 begins with a character not in string2, strspn returns 0.

Return Value

The strspn function returns an integer value specifying the length of the segment in string1 consisting entirely of characters in string2.

See Also

strcspn

Example

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

char *string = "cabbage";
int result;

main()
{
    result = strspn(string, "abc");  /* result = 5 */
    printf("The string starting with \"abc\" is %d bytes long.", result);
}
strspn

This program uses `strspn` to determine the length of the segment in the string "cabbage" consisting of a's, b's, and c's.
# Summary

```c
#include <string.h>  // Required only for function declarations

char *strstr(string1, string2);
const char *string1;  // Searched string
const char *string2;  // String to search for
```

## Description

The `strstr` function returns a pointer to the first occurrence of `string2` in `string1`.

## Return Value

The `strstr` function returns a pointer to the first occurrence of `string2` in `string1`, or `NULL` if it does not find `string2` in `string1`.

## See Also

`strcspn`

## Example

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

main()
{
    char *string1 = "needle in a haystack";
    char *string2 = "hay";

    printf("%s\n", strstr(string1, string2));
}
```

Output:

```
haystack
```
**strstr**

This program uses **strstr** to return a pointer to the first location of hay in the string **string1**, then prints the remainder of the string.
# Summary

```c
#include <stdlib.h>

double ** double strtod(nptr, endptr);
const char ** nptr;
char ** endptr;

long strtol(nptr, endptr, base);
const char ** nptr;
char ** endptr;
int base;

unsigned long int strtoul(nptr, endptr, base);
const char ** nptr;
char ** endptr;
int base;
```

## Description

The **strtod**, **strtol**, and **strtoul** functions convert a character string to a double-precision value, a long-integer value, or an unsigned long integer value, respectively. The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. These functions stop reading the string at the first character they cannot recognize as part of a number (which may be the null character at the end of the string); with **strtol** or **strtoul** this terminating character could also be the first numeric character greater than or equal to the **base**. If **endptr** is not NULL, it points to the character that stopped the scan.

The **strtod** function expects **nptr** to point to a string with the following form:

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

The first character that doesn’t fit this form stops the scan.
## strtod – strtoul

The **strtol** function expects `nptr` to point to a string with the following form:

```
[whitespace] [sign] [0] [{ x | X }] [digits]
```

The **strtoul** function expects `nptr` to point to a string of this form:

```
[whitespace] [0] [{ x | X }] [digits]
```

If `base` is between 2 and 36, then it is used as the base of the number. If `base` is 0, the initial characters of the string pointed to by `nptr` are used to determine the base: if the first character is '0' and the second character is a digit '0' – '7', then the string is interpreted as an octal integer; if the first character is '0' and the second character is 'x' or 'X', then the string is interpreted as a hexadecimal integer; if the first character is '1' – '9', then the string is interpreted as a decimal integer. The letters from `a` through `z` (or `A` through `Z`) are assigned the values 10 through 35; only letters whose assigned value are less than `base` are permitted.

### Return Value

The **strtod** function returns the value of the floating-point number, except when the representation would cause an overflow or underflow, in which case it returns `±HUGE_VAL`. The function returns zero if no conversion could be performed or an underflow occurred.

The **strtol** function returns the value represented in the string, except when the representation would cause an overflow or underflow, in which case it returns `LONG_MAX` or `LONG_MIN`. The functions returns zero if no conversion could be performed.

The **strtoul** function returns the converted value, if any. If no conversion can be performed, the function returns zero. The function returns `ULONG_MAX` on overflow.

In all three functions **errno** is set to **ERANGE** if overflow or underflow would occur.
See Also
atof, atol

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 = %sn",string);
    printf("  strtod = %f
",x);
    printf("  Stopped scan at %s\n", stopstring);

    string = "10110134932";
    printf("string = %sn",string);
    for (bs = 2; bs <= 8; bs *= 2) {
        l = strtoul(string,&stopstring,bs);
        printf("  strtoul = %ld (base %d)\n", l, bs);
        printf("  Stopped scan at %s\n", stopstring);
    }
}

Output:

string = 3.1415926This stopped it
    strtod = 3.141593
        Stopped scan at This stopped it

string = 10110134932
    strtoul = 45 (base 2)
        Stopped scan at 34932

    strtoul = 4423 (base 4)
        Stopped scan at 4932

    strtoul = 2134108 (base 8)
        Stopped scan at 932
strtol – strtoul

This program uses \texttt{strtol} and \texttt{strtoul} to convert two strings to a double-precision and a long-integer value, respectively.
# Summary

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

Required only for function declarations

```
char *strtok(string1, string2);
char *string1;
char *string2;
```

Find token in `string1`  
String containing token(s)  
Set of delimiter characters

## Description

The `strtok` function reads `string1` as a series of zero or more tokens and `string2` as the set of characters serving as delimiters of the tokens in `string1`. The tokens in `string1` may be separated by one or more of the delimiters from `string2`. The tokens are broken out of `string1` by a series of calls to `strtok`.

In the first call to `strtok` for a given `string1`, `strtok` searches for the first token in `string1`, skipping leading delimiters. A pointer to the first token is returned.

To read the next token from `string1`, call `strtok` with a NULL value for the `string1` argument. The NULL `string1` argument causes `strtok` to search for the next token in the previous token string. The set of delimiters may vary from call to call, so `string2` can take any value.

---

**Note**

Calls to `strtok` will modify `string1`, since each time `strtok` is called it inserts a null value ('\0') after the token in `string1`.

---

## Return Value

The first time `strtok` is called, it returns a pointer to the first token in `string1`. In later calls with the same token string, `strtok` returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null terminated.
strtok

• See Also
strcspn, strspn

• Example

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

char *string = "a string, of ,, tokens ";
char *token;

main()
{
  /* Establish string and get the first */
  /* token:
  token = strtok(string," ","");
  while (token != NULL)
  /* While there are tokens in "string" */
  /* tokens in "string" */
  {
    printf("The token is: %s\n", token);
    /* Get next token: */
    token = strtok(NULL," ","");
  }
}

In this program, a loop uses **strtok** to print all the tokens (separated by commas or blanks) in the string named **string**.
### Summary

```c
#include <string.h>
```

Required only for function declarations.

```c
char *strupr(string);
```

String to be capitalized.

### Description

The `strupr` function converts any lowercase letters in the given `string` to uppercase. Other characters are not affected.

### Return Value

The `strupr` function returns a pointer to the converted `string`. There is no error return.

### See Also

`strlwr`

### Example

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

char string[100] = "This Was a Mixed-Case String", *copy;

main()
{
    copy = strupr(strdup(string));
    printf("The result string is: %s", copy);
}
```

This program duplicates a string named `string` and uses `strupr` to convert all lowercase letters in the copy to uppercase.
swab

- **Summary**

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

void swab(source, destination, n);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>Data to be copied and swapped</td>
</tr>
<tr>
<td>destination</td>
<td>Storage location for swapped data</td>
</tr>
<tr>
<td>n</td>
<td>Number of bytes copied</td>
</tr>
</tbody>
</table>

- **Description**

The `swab` function copies `n` bytes from `source`, swaps each pair of adjacent bytes, and stores the result at `destination`. The integer `n` should be an even number to allow for swapping. The `swab` function is typically used to prepare binary data for transfer to a machine that uses a different byte order.

- **Return Value**

There is no return value.

- **See Also**

`fgetc`, `fputc`

- **Example**

```c
#define NBYTES 1024
char from[NBYTES], to[NBYTES];

main()
{
    strcpy(from, "badcfeghjiklnmopqr");
    strcpy(to, "................");
    printf("%s %s\n", from, to);

    swab(from, to, NBYTES);    /* to = "abcdefghijklmnopqr" */
    printf("%s %s\n", from, to);
}
```

492
This program uses `swab` to copy the string named `from` to the string named `to` and swap each adjacent pair of bytes.
system

Summary

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

int system(string);
const char *string;
```

Required only for function declarations
For ANSI compatibility

Command to be executed

Description

The system function passes the given string to the command interpreter and executes the string as an DOS command. The system function refers to the COMSPEC and PATH environment variables to locate the DOS file COMMAND.COM, which is used to execute the string command.

Return Value

The system function returns the value 0 if string is successfully executed. A return value of −1 indicates an error, and errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>The argument list for the command exceeds 128 bytes, or the space required for the environment information exceeds 32K.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>COMMAND.COM cannot be found.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>The COMMAND.COM file has an invalid format and is not executable.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.</td>
</tr>
</tbody>
</table>
See Also

excl, execle, execvp, execv, execve, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

Example

#include <stdio.h>

int result;

main()
{
    /* Place version number in "result.log": */
    result = system("ver >>result.log");

    /* Type "result.log" to the screen: */
    result = system("type result.log");
}

This program uses system to place the DOS version number in a file named result.log and then displays result.log on the screen.
■ Summary

#include <math.h>

double tan(x); Calculate tangent of x
double tanh(x); Calculate hyperbolic tangent of x
double x; Radians

■ Description

The tan and tanh functions return the tangent and hyperbolic tangent of x, respectively.

■ Return Value

The tan function returns the tangent of x. If x is large, a partial loss of significance in the result may occur. 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.

The tanh function returns the hyperbolic tangent of x. There is no error return.

■ See Also

acos, asin, atan, atan2, cos, cosh, sin, sinh

■ Example

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

main()
{
  double pi = 3.1415926535, x,y;
  pi = 3.1415926535;
\textbf{tan – tanh}

\begin{verbatim}

x = tan(pi/4);              /* x is 1.0 */
printf("The tan(%f) = %f\n",pi/4,x);

y = tanh(x);               /* y is 0.761594 */
printf("The tanh(%f) = %f\n",x,y);

\end{verbatim}

This program displays the value of the tangent of $\pi/4$ and the hyperbolic tangent of 1.0.
# Summary

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

long tell(handle);  
Handle referring to open file

# Description

The `tell` function gets the current position of the file pointer (if any) associated with `handle`. The position is expressed as the number of bytes from the beginning of the file.

# Return Value

The `tell` function returns the current position. A return value of -1L indicates an error, and `errno` is set to `EBADF` to indicate an invalid file-handle argument. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

# See Also

`ftell`, `lseek`

# Example

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

int fh;
long position;

main()
{
    fh = open("data",O_RDONLY);

    /* Report position at start of file: */
    position = tell(fh);             /* Position = 0 */
    printf("position = %ld\n", position);

    /* Position pointer at end -3 of file: */
```
tell

lseek(fh, -3L, SEEK_END);
position = tell(fh);    /* Position = file length -3 */

printf("position = %ld\n", position);

/* Assume pointer was repositioned here */
lseek(fh, position, SEEK_SET);    /* Put pointer back */
}

/* at previous position */

This program uses tell to calculate the beginning and the position 3 bytes from the end of the file named data.
# Summary

```c
#include <stdio.h>

char *tmpnam(string);
char *string;

char *tempnam(dir, prefix);
char *dir;
char *prefix;
```

## Description

The `tmpnam` function generates a temporary file name that can be used as a temporary file. This name is stored in `string`. If `string` is `NULL`, then `tmpnam` leaves the result in an internal static buffer. Thus, any subsequent calls will destroy this value. If `string` is not `NULL`, it is assumed to point to an array of at least `L_tmpnam` bytes, where the value of `L_tmpnam` is defined in the `stdio.h` include file. The `tmpnam` function uses `malloc` to allocate space for the file name; the user is responsible for freeing this space when it is no longer needed.

The character string that `tmpnam` creates consists of the path prefix defined by the `P_tmpdir` entry in `stdio.h`, followed by a sequence consisting of the digit characters '0' through '9'; the numerical value of this string can range from 1 to 65535. Changing the definitions of `L_tmpnam` or `P_tmpdir` in `stdio.h` does not change the operation of `tmpnam`.

The `tempnam` function allows the user to create a temporary file in another directory. The `prefix` is the prefix to the file name. The `tempnam` function looks for the file with the given name in the following directories, listed in order of precedence:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Directory Used by tempnam</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP environment variable is set, and directory specified by TMP exists.</td>
<td>Directory specified by TMP</td>
</tr>
<tr>
<td>TMP environment variable not set, or directory specified by TMP does not exist.</td>
<td>The <code>dir</code> argument to <code>tempnam</code></td>
</tr>
</tbody>
</table>
The dir argument is NULL, or dir is name of non-existent directory.

P_tmpdir does not exist.

The current working directory

If all this fails, tempnam returns the value NULL.

Return Value

The tmpnam and tempnam functions both return a pointer to the name generated, unless it is impossible to create this name, or the name is not unique. If the name cannot be created or if it already exists, tmpnam and tempnam return the value NULL.

See Also

tmpfile

Example

#include <stdio.h>

main()
{
    char *name1, *name2;

    /* Create a temporary file name for */
    /* the current working directory: */
    if ((name1 = tmpnam(NULL)) != NULL)
        printf("%s is safe to use as a temporary file.\n", name1);
    else
        printf("Cannot create a unique file name\n");

    /* Create a temporary file name for */
    /* directory "a:mp" with the prefix "stq":
    if ((name2 = tempnam("a:tmp","stq")) != NULL)
        printf("%s is safe to use as a temporary file.\n", name2);
    else
        printf("Cannot create a unique file name\n");
}
This program uses `tempnam` to create two temporary file names: one in the current working directory, and one in `A:\TMP` (with a prefix of `stq`.)
time

- Summary

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

- Description

The `time` function returns the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970, according to the system clock.

---

**Note**

The value returned by `time` assumes that the system time is GMT. Therefore, to get a true time stamp, you must adjust this by the value of `timezone`, a global variable set by the `tzset` function.

---

The return value is also stored in the location given by `timeptr`; `timeptr` may be `NULL`, in which case the return value is not stored.

- Return Value

The `time` function returns the time in elapsed seconds. There is no error return.

- See Also

`asctime`, `ftime`, `gmtime`, `localtime`, `utime`

- Example

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

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

This program uses time to obtain the current time in long-integer form, then displays this time.
tmpfile

- **Summary**

```c
#include <stdio.h>

FILE *tmpfile(); // Pointer to file structure
```

- **Description**

The `tmpfile` function creates a temporary file and returns a pointer to that file. If the file cannot be opened, `tmpfile` returns a `NULL` pointer.

This temporary file is automatically deleted when the program terminates normally, or when `rmtmp` is called, assuming that the current working directory does not change. The temporary file is opened in "w+" mode.

- **Return value**

The `tmpfile` function returns a stream pointer, unless it cannot open the file, in which case it returns a `NULL` pointer.

- **See Also**

tmpnam, tempnam, rmtmp

- **Example**

```c
#include <stdio.h>

FILE *stream;
char tempstring[] = "String to be temporarily written";

main()
{
    if ((stream = tmpfile()) == NULL) /* Create temporary file */
        perror("Could not open new temporary file");
    else
    {
        fprintf(stream, "%s", tempstring);
        printf("Temporary file was created, and ");
    }
    rmtmp(); /* Remove temporary file */
}
```

506
This program uses `tmpfile` to create a temporary file, then deletes this file.
toascii — _toupper

■ Summary

```c
#include <ctype.h>

int toascii(c);        // Convert c to ASCII character
int tolower(c);        // Convert c to lowercase if appropriate
int _tolower(c);       // Convert c to lowercase
int toupper(c);        // Convert c to uppercase if appropriate
int _toupper(c);       // Convert c to uppercase
int c;                // Character to be converted
```

■ Description

The `toascii`, `tolower`, `_tolower`, `toupper`, and `_toupper` macros convert a single character as specified.

The `toascii` macro sets all but the low-order 7 bits of `c` to 0, so that the converted value represents a character in the ASCII character set. If `c` already represents an ASCII character, `c` is unchanged.

The `tolower` macro converts `c` to lowercase if `c` represents an uppercase letter. Otherwise, `c` is unchanged.

The `_tolower` macro is a version of `tolower` to be used only when `c` is known to be uppercase. The result of `_tolower` is undefined if `c` is not an uppercase letter.

The `toupper` macro converts `c` to uppercase if `c` represents a lowercase letter. Otherwise, `c` is unchanged.

The `_toupper` macro is a version of `toupper` to be used only when `c` is known to be lowercase. The result of `_toupper` is undefined if `c` is not a lowercase letter.
toascii, _toupper

- Return Value

The toascii, tolower, _tolower, toupper, and _toupper macros return the possibly converted character c. There is no error return.

- See Also

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

---

Note

These routines are implemented as macros. However, tolower and toupper are also implemented as functions, because the macro versions do not correctly handle arguments with side effects. The function versions can be used by removing the macro definitions through #undef directives or by not including ctype.h. Function declarations of tolower and toupper are given in stdlib.h.

---

- Example

#include <stdio.h>
#include <ctype.h>

int ch;

main()
{
    for ( ch = 0; ch <= 0x7f; ch++ )
    {
        /* Apply "toupper" and "tolower": */
        printf(" toupper =%#04x", toupper(ch));
        printf(" tolower =%#04x", tolower(ch));

        /* Apply "_toupper": */
        if (islower(ch))
            printf(" _toupper =%#04x", _toupper(ch));

        /* Apply "_tolower": */
        if (isupper(ch))
            printf(" _tolower =%#04x", _tolower(ch));
    }
This program uses \texttt{toupper} and \texttt{tolower} to analyze all characters between 0x0 and 0x7F. It also applies \texttt{-toupper} and \texttt{-tolower} to any code in this range for which these functions make sense.
tzset

### Summary

```c
#include <time.h>
void tzset();
```

- `int daylight;`: Daylight saving time flag
- `long timezone;`: Difference in seconds from GMT

### Description

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 three-letter time-zone name, such as PST, followed by an optionally signed number giving the difference in hours between Greenwich mean time and local time. The number may be followed by a three-letter daylight saving time zone, such as PDT. For example, “PST8PDT” represents a valid `TZ` value for the Pacific time zone.

The following values are assigned to the variables `daylight`, `timezone`, and `tzname` when `tzset` is called:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>timezone</code></td>
<td>The difference in seconds between Greenwich mean time and local time</td>
</tr>
<tr>
<td><code>daylight</code></td>
<td>Nonzero value if a daylight saving time zone is specified in the <code>TZ</code> setting; otherwise, 0</td>
</tr>
<tr>
<td><code>tzname[0]</code></td>
<td>The string value of the three-letter time-zone name from the <code>TZ</code> setting</td>
</tr>
<tr>
<td><code>tzname[1]</code></td>
<td>The string value of the daylight saving time zone, or an empty string if the daylight saving time zone is omitted from the <code>TZ</code> setting</td>
</tr>
</tbody>
</table>
If `TZ` is not currently set, the default is “PST8PDT”, which corresponds to the Pacific time zone. The default for `daylight` is 1; for `timezone`, 28800; for `tzname[0]`, “PST”; and for `tzname[1]`, “PDT”.

- **Return Value**

There is no return value.

- **See Also**

  `asctime`, `ftime`, `localtime`

- **Example**

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

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

main()
{
    putenv("TZ=EST5");
    tzset();

    /* daylight = 0 */
    printf("daylight = %d\n", daylight);

    /* timezone = 18000 */
    printf("timezone = %ld\n", timezone);

    /* tzname[0] = "EST" */
    printf("tzname[0] = %s\n", tzname[0]);
}
```

This program first sets up the time zone by placing the variable named `TZ=EST5` in the environment table. It then uses `tzset` to set the variables named `daylight`, `timezone`, and `tzname`.
ultoa

■ Summary

```c
#include <stdlib.h>

char ultoa(value, string, radix);
unsigned long value;
char *string;
int radix;
```

■ Description

The `ultoa` function converts the digits of the given `value` to a null-terminated 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 2–36.

■ Return Value

The `ultoa` function returns a pointer to `string`. There is no error return.

■ See Also

`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.

---

■ Example

```c
#include <stdlib.h>

int radix = 16;
char buffer[40];
char *p;

main()
```
This program converts the long integer 1344115000 to a string and displays that string.
umask

- Summary

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
```

```c
int umask(pmode);
int pmode;
```

Required only for function declarations

- Description

The `umask` function sets the file-permission mask of the current process to the mode specified by `pmode`. The file-permission mask is used to modify the permission setting of new files created by `creat`, `open`, or `sopen`. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The argument `pmode` is a constant expression containing one or both of the manifest constants `S_IWWRITE` and `S_IRREAD`, 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S_IWWRITE</code></td>
<td>Writing not allowed (file is read only)</td>
</tr>
<tr>
<td><code>S_IRREAD</code></td>
<td>Reading not allowed (file is write only)</td>
</tr>
</tbody>
</table>

For example, if the write bit is set in the mask, any new files will be read only.

---

**Note**

Under DOS, all files are readable—it is not possible to give write-only permission. Therefore, setting the read bit with `umask` has no effect on the file's permissions.
- Return Value

The `umask` function returns the previous value of `pmode`. There is no error return.

- See Also

`chmod`, `creat`, `mkdir`, `open`

- Example

```c
#include <systypes.h>
#include <sysat.h>
#include <io.h>
#include <stdio.h>

int oldmask;

main()
{
    /* Create read-only files: */
    oldmask = umask(S_IWRITE);
    printf("oldmask =%#x\n", oldmask);
}
```

This program uses `umask` to set the file-permission mask so that all future files will be created as read-only files. It also displays the old mask.
ungetc

- Summary

```c
#include <stdio.h>

int ungetc(c, stream);
```

- Description

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` onto the stream using `ungetc` is ignored. The `ungetc` function returns an error value if nothing has yet been read from `stream` or if `c` cannot be pushed back.

Characters placed on the stream by `ungetc` may be erased if an `fseek` or `rewind` function is called before the character is read from the `stream`.

- Return Value

The `ungetc` function returns the character argument `c`. The return value `EOF` indicates a failure to push back the specified character.

- See Also

`getc`, `getchar`, `putc`, `putchar`
Example

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

FILE *stream;
int ch;
int result = 0;

main()
{
    stream = stdin;
    printf("Input an integer: ");

    /* Read in and convert number: */
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';

    if (ch != EOF)
        ungetc(ch, stream);  /* Put non-digit back */

    printf("Number = %d\nNext character in stream = \"%c\"\n", result, getc(stream));
}
```

This program first converts a character representation of an unsigned integer to an integer. If the program encounters a character that is not a digit, the program uses `ungetc` to replace it in the stream.
ungetch

- Summary

```c
#include <conio.h>

int ungetch(c);
int c;
```
Required only for function declarations

- Description

The `ungetch` function pushes the character `c` back to the console, causing `c` to be the next character read. The `ungetch` function fails if it is called more than once before the next read.

- Return Value

The `ungetch` function returns the character `c` if it is successful. A return value of `EOF` indicates an error.

- See Also

cscanf, getch, getche
Example

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

char buffer[100];
int count = 0;
int ch;

main()
{
    ch = getche();
    while (isspace(ch))       /* skip preceding white space */
        ch = getche();
    while (count < 99)       /* Gather token */
    {
        if (isspace(ch))       /* End of token */
            break;
        buffer[count++] = ch;
        ch = getche();
    }
    ungetch(ch);            /* Put back delimiter */
    buffer[count] = '\0';  /* Null terminate the token */
    printf("\ntoken = %s\n", buffer);
}
```

In this program, tokens are read from the keyboard delimited by blanks and new-line characters. When the program encounters a delimiter, it uses `ungetch` to replace the delimiter in the keyboard buffer.
**unlink**

- **Summary**

```c
#include <io.h>   // Required only for function declarations
#include <stdio.h> // Use either io.h or stdio.h

int unlink(pathname);
char *pathname;   // Path name of file to be removed
```

- **Description**

The `unlink` function deletes the file specified by `pathname`.

- **Return Value**

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCESS</td>
<td>Path name specifies a directory or a read-only file</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found</td>
</tr>
</tbody>
</table>

- **See Also**

close, remove

- **Example**

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

int result;

main()
{
    result = unlink("tmpfile");
    if (result == -1)
        perror("Couldn't delete tmpfile");
    else
```
This program uses `unlink` to delete a file named `TMPFILE`.
int utime(pathname, times);

char *pathname;  // File path name
struct utimbuf *times;  // Pointer to stored time values

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Path name specifies directory or read-only file</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files (the file must be opened to change its modification time)</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or path name not found</td>
</tr>
</tbody>
</table>
- See Also

asctime, ctime, fstat, ftime, gmtime, localtime, stat, time

- Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utime.h>

main()
{
    int savestderr;

    if (utime("/tmp/data", NULL) == -1)
        perror("utime failed");
    else
        printf("File time modified.");
}
```

This program uses utime to set the file-modification time to the current time.
**Summary**

```c
#include <stdarg.h>
#include <varargs.h>
#include <stdio.h>

void va_start(arg-ptr);

void va_start(arg-ptr, prev-param);

type va_arg(arg-ptr, type);

void va_end(arg-ptr);

va_list arg-ptr;
type prev-param
va alist
va dcl
```

**Description**

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

Both versions of the macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be accessed through the parameter names. The
optional arguments are accessed through the `stdarg.h` or `varargs.h` macros, which set a pointer to the first optional argument in the argument list, retrieve arguments from the list, and reset the pointer when argument processing is completed.

The proposed ANSI C standard macros, defined in `stdarg.h`, are used as follows:

1. All required arguments to the function are declared as parameters in the usual way. The `va_dcl` macro is not used with the `stdarg.h` macros.

2. The `va_start` macro sets `arg-ptr` to the first optional argument in the list of arguments passed to the function. The argument `arg-ptr` must have `va_list` type. The argument `prev-param` is the name of the required parameter immediately preceding the first optional argument in the argument list. The `va_start` macro must be used before `va_arg` is used for the first time.

3. The `va_arg` macro does the following:
   - Retrieves a value of the given `type` from the location given by `arg-ptr`
   - Increments `arg-ptr` to point to the next argument in the list, using the size of `type` to determine where the next argument starts

   The `va_arg` macro can be used any number of times within the function to retrieve arguments from the list.

4. After all arguments have been retrieved, `va_end` resets the pointer to `NULL`.

The UNIX System V macros, defined in `varargs.h`, operate in a slightly different manner, as follows:

1. Any required arguments to the function can be declared as parameters in the usual way.

2. The last (or only) parameter to the function represents the list of optional arguments. This parameter must be named `va_alist` (not to be confused with `va_list`, which is defined as the type of `va_alist`).

3. The `va_dcl` macro appears after the function definition and before the opening left brace of the function. This macro is defined as a complete declaration of the `va_alist` parameter, including the terminating semicolon; therefore, no semicolon should follow `va_dcl`. 
va_arg – va_start

4. Within the function, the \texttt{va_start} macro sets \texttt{arg-ptr} to the beginning of the list of optional arguments passed to the function. The \texttt{va_start} macro must be used before \texttt{va_arg} is used for the first time. The argument \texttt{arg-ptr} must have \texttt{va_list} type.

5. The \texttt{va_arg} macro does the following:
   - Retrieves a value of the given \texttt{type} from the location given by \texttt{arg-ptr}
   - Increments \texttt{arg-ptr} to point to the next argument in the list, using the size of \texttt{type} to determine where the next argument starts

The \texttt{va_arg} macro can be used any number of times within the function to retrieve the arguments from the list.

6. After all arguments have been retrieved, \texttt{va_end} resets the pointer to \texttt{NULL}.

The proposed ANSI C standard macros, defined in \texttt{stdarg.h}, operate in a slightly different manner, as follows:

1. All required arguments to the function are declared as parameters in the usual way. The \texttt{va_dcl} macro is not used with the \texttt{stdarg.h} macros.

2. The \texttt{va_start} macro sets \texttt{arg-ptr} to the first optional argument in the list of arguments passed to the function. The argument \texttt{arg-ptr} must have \texttt{va_list} type. The argument \texttt{prev-param} is the name of the required parameter immediately preceding the first optional argument in the argument list. The \texttt{va_start} macro must be used before \texttt{va_arg} is used for the first time.

3. The \texttt{va_arg} macro does the following:
   - Retrieves a value of the given \texttt{type} from the location given by \texttt{arg-ptr}
   - Increments \texttt{arg-ptr} to point to the next argument in the list, using the size of \texttt{type} to determine where the next argument starts

The \texttt{va_arg} macro can be used any number of times within the function to retrieve arguments from the list.

4. After all arguments have been retrieved, \texttt{va_end} resets the pointer to \texttt{NULL}.  

526
■ Return Value

The `va_arg` macro returns the current argument; `va_start` and `va_end` do not return values.

■ See Also

`vfprintf`, `vprintf`, `vssprintf`

■ Example

```c
/*************************** Example 1 ***************************/
#include <stdio.h>
#include <stdarg.h>

main()
{
  int n;
  n = average(2, 3, 4, -1); /* Call with 4 arguments */
  printf("Average is: %d\n", n);
  n = average(5, 7, 9, 11, -1); /* Call with 5 arguments */
  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_marker, int)) >= 0; sum+=i, count++)
    return (sum/count);
}
```
va_arg – va_start

Example 1 demonstrates how to pass a variable number of arguments using the ANSI C standard.

/****************************Example 2*****************************/
#include <stdio.h>
#include <varargs.h>

main()
{
    int n;
    n = average(2, 3, 4, -1);              /* Call with 4 arguments */
    printf("Average is: %d\n", n);         /* -1 terminates the list */

    n = average(5, 7, 9, 11, -1);         /* Call with 5 arguments */
    printf("Average is: %d\n", n);         /* -1 terminates the list */
}

average(va_alist)
va_dcl
{
    int i = 0, count = 0, sum = 0;
    va_list arg_marker;

    va_start(arg_marker);
    va_start(arg_marker);

    for (; (i = va_arg(arg_marker, int)) >= 0; sum+=i, count++);

    return(count ?(sum/count) : count);
}

Example 2 shows Example 1 rewritten for compatibility with the UNIX System V standard.
vfprintf - vsprintf

- Summary

```c
#include <stdio.h>
#include <varargs.h>  \* Required for compatibility with UNIX V
#include <stdarg.h>   \* Required for compatibility with proposed ANSI C standard
```

```c
int vfprintf(FILE *stream, const char *format-string, va_list arg-ptr);
int vprintf(const char *format-string, va_list arg-ptr);
int vsprintf(char *buffer, const char *format-string, va_list arg-ptr);
```

- Description

The `vfprintf`, `vprintf`, and `vsprintf` functions format and output data to `stream`, the standard output, or `buffer`, respectively. These functions are similar to their counterparts `fprintf`, `printf`, and `sprintf`, but `vfprintf`, `vprintf`, and `vsprintf` accept a pointer to a list of arguments rather than a list of arguments.

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`.

The `arg-ptr` parameter has type `va_list`, which is defined in `varargs.h` and `stdarg.h`. The `arg-ptr` parameter points to a list of arguments that are converted and output according to the corresponding format specifications in the `format-string`.

- Return Value

The return value is the number of characters written.
vfprintf - vsprintf

- See also

fprintf, printf, sprintf, va_arg, va_end, va_start

- Example

/****************************Example 1***************************/
#include <stdio.h>
#include <varargs.h>

main()
{
    int line = 1;
    char *filename = "EXAMPLE";

    /* Call "error" with a format */
    /* string and two parameters */
    error("Error: line %d, file " filename);

    /* Call "error" with just a format */
    error("Syntax error\n"); /* string. */
}

error(va_alist)
va_dcl
{
    char *fmt;
    va_list arg_ptr;

    /* "arg_ptr" points to format string */
    va_start(arg_ptr);

    /* "arg_ptr" points to first argument */
    fmt = va_arg(arg_ptr, char *);
    vprintf(fmt, arg_ptr);
    va_end(arg_ptr);
}

Output:

Error: line 1, file EXAMPLE
Syntax error
Example 1 conforms to the UNIX System V standard. It uses `vprintf` to set up an error routine that takes a variable number of arguments and displays the appropriate error messages.

```c
/****************** Example 2 ******************/
#include <stdio.h>
#include <stdarg.h>

main()
{
    int line = 1;
    char *filename = "EXAMPLE";
    /* Call "error" with a format */
    /* string and two parameters */
    error("Error: line %d, file %s\n", line, filename);
    /* Call "error" with just */
    error("Syntax error\n"); /* a format string. */
}

error(fmt)
char *fmt;
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt); /* "arg_ptr" points to */
    vprintf(fmt, arg_ptr);
    va_end(arg_ptr);
}

Output:

Error: line 1, file EXAMPLE
Syntax error

Example 2 shows Example 1 rewritten to conform to the ANSI C standard.
write

- Summary

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

int write(handle, buffer, count);
int handle;
char *buffer;
unsigned int count;

- Description

The write function writes count bytes from buffer into the file associated with handle. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer (if any) is increased by the number of bytes actually written.

- Return Value

The write function returns the number of bytes actually written. The return value may be positive but less than count (for example, when running out of space on a disk before count bytes are written).

A return value of -1 indicates an error. In this case, errno is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file handle or file not opened for writing</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>No space left on device</td>
</tr>
</tbody>
</table>

If you are writing more than 32K (the maximum size for type int) to a file, the return value should be of type unsigned int. (See the example that follows.) However, the maximum number of bytes that can be written to a file is 65,534 at one time, since 65,535 (or 0xFFFF) is indistinguishable from -1, and so would return 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.
See Also

fwrite, open, read

Note

When writing to files opened in text mode, a CONTROL-Z character is treated as the logical end-of-file. When writing to a device, a CONTROL-Z character in the buffer causes output to be terminated.

Example

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

char buffer[6000] = "This is a test of 'write' function";

main()
{
    int fh;
    unsigned int nbytes = 60000, byteswritten;

    if ((fh = open("c:/data/conf.dat",O_WRONLY)) == -1)
    {
        perror("Open failed on output file");
        exit(1);
    }

    if ((byteswritten = write(fh, buffer, nbytes)) == -1)
        perror(""");
    else
        printf("Wrote %u bytes to file\n", byteswritten);
}
```

This program opens a file for output and uses write to write 60,000 bytes to the file.
Appendixes

Error Messages 535
Common Libraries for DOS, XENIX, UNIX, and ANSI 541
Appendix A

Error Messages

A.1 Introduction 537
A.2 errno Values 537
A.3 Math Errors 540
A.1 Introduction

This appendix lists and describes the values to which the `errno` variable can be set when an error occurs in a call to a library routine. Note that only some routines set the `errno` variable. The reference pages for the routines that set `errno` upon error explicitly mention the `errno` variable. (The reference pages are located in Part 2 of this manual.) If no mention of `errno` occurs, the routine does not set `errno`.

An error message is associated with each `errno` value. This message, along with a user-supplied message, can be printed by using the `perror` function.

The value of `errno` reflects the error value for the last call that set `errno`. The `errno` value is not automatically cleared by later successful calls. Thus, to obtain accurate results, you should test for errors and print error messages, if desired, immediately after a call.

The include file `errno.h` contains the definitions of the `errno` values. However, not all of the definitions given in `errno.h` are used under DOS. The full set of values is provided in the include file to maintain compatibility with the XENIX and UNIX include files having the same name.

This appendix lists only the `errno` values used under DOS. For the complete listing of `errno` values, see the `errno.h` include file.

Also listed in this appendix are the errors produced by math routines when an error occurs. These errors correspond to the exception types defined in `math.h` and returned by the `matherr` function when a math error occurs.

A.2 `errno` Values

Table A.1 gives the `errno` values used on DOS, the system error message corresponding to each value, and a brief description of the circumstances that cause the error.
### Table A.1

**errno Values and Their Meanings**

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>Arg list too long.</td>
<td>The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K bytes.</td>
</tr>
<tr>
<td>EACCES</td>
<td>Permission denied.</td>
<td>Access denied: the file’s permission setting does not allow the specified access. This error can occur in a variety of circumstances; it signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file’s attributes. For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under DOS 3.0 and later, EACCES may also indicate a locking or sharing violation. The error can also occur in an attempt to rename a file or directory or to remove an existing directory.</td>
</tr>
<tr>
<td>EBADF</td>
<td>Bad file number.</td>
<td>The specified file handle is not a valid file-handle value or does not refer to an open file; or an attempt was made to write to a file or device opened for read-only access (or vice versa).</td>
</tr>
<tr>
<td>EDEADLOCK</td>
<td>Resource deadlock would occur.</td>
<td>Locking violation: the file cannot be locked after 10 attempts (DOS Version 3.0 and later only).</td>
</tr>
<tr>
<td>EDOM</td>
<td>Math argument.</td>
<td>The argument to a math function is not in the domain of the function.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>File exists.</td>
<td>The O_CREAT and O_EXCL flags are specified when opening a file, but the named file already exists.</td>
</tr>
</tbody>
</table>
## Table A.1 (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Invalid argument.</td>
<td>An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files.</td>
<td>No more file handles are available, so no more files can be opened.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No such file or directory.</td>
<td>The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path name does not specify an existing directory.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>Exec format error.</td>
<td>An attempt is made to execute a file that is not executable or that has an invalid executable file format.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough core.</td>
<td>Not enough memory is available. This message can occur when insufficient memory is available to execute a child process or when the allocation request in an <code>sbrk</code> or <code>getcwd</code> call cannot be satisfied.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>No space left on device.</td>
<td>No more space for writing is available on the device (for example, the disk is full).</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Result too large.</td>
<td>An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the path-name argument to the <code>getcwd</code> function is longer than expected).</td>
</tr>
<tr>
<td>EXDEV</td>
<td>Cross-device link.</td>
<td>An attempt was made to move a file to a different device (using the <code>rename</code> function).</td>
</tr>
</tbody>
</table>
A.3 Math Errors

The following errors can be generated by the math routines of the C runtime library. These errors correspond to the exception types defined in `math.h` and returned by the `matherr` function when a math error occurs; see the `matherr` reference page in Part 2 of this manual for details.

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMAIN</td>
<td>An argument to the function is outside the domain of the function.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>The result is too large to be represented in the function’s return type.</td>
</tr>
<tr>
<td>PLOSS</td>
<td>A partial loss of significance occurred.</td>
</tr>
<tr>
<td>SING</td>
<td>Argument singularity: an argument to the function has an illegal value (for example, passing the value 0 to a function that requires a nonzero value).</td>
</tr>
<tr>
<td>TLOSS</td>
<td>A total loss of significance occurred.</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>The result is too small to be represented. (This condition is not currently supported.)</td>
</tr>
</tbody>
</table>
Appendix B
Common Libraries for
DOS, XENIX, UNIX, and ANSI

B.1 Introduction 543
B.2 Run-Time Routines 543
B.2.1 Routines Common to DOS and XENIX 543
B.2.2 Routines Common to DOS and UNIX System V 544
B.2.3 Routines Specific to DOS 545
B.2.4 ANSI Extensions 545
B.3 Global Variables 547
B.3.1 Variables Common to DOS and XENIX 547
B.3.2 Variables Common to DOS and UNIX System V 547
B.3.3 Variables Specific to DOS 548
B.4 Include Files 548
B.4.1 Include Files Common to DOS and XENIX 548
B.4.2 Include Files Common to DOS and UNIX System V 549
B.4.3 Include Files Specific to DOS 549
B.4.4 ANSI Include Files 549
B.5 Differences Between Routines Common to DOS and XENIX 550
B.5.1 abort 550
B.5.2 access 550
B.5.3 atexit 551
B.5.4 chdir 551
B.5.5 chmod 551
B.5.6 creat 552
B.5.7 exec 552
B.5.8 fopen, freopen 553
B.5.9 fread 553
B.5.10 fseek 554
B.5.11 fstat 554
B.5.12 ftell 555
B.5.13 ftime 555
B.5.14 fwrite 555
B.5.15 getpid 555
B.5.16 locking 556
B.5.17 lseek 556
B.5.18 open 556
B.5.19 read 557
B.5.20 signal 557
B.5.21 stat 557
B.5.22 system 558
B.5.23 umask 559
B.5.24 unlink 559
B.5.25 utime 559
B.5.26 write 559
B.1 Introduction

This appendix lists and describes routines from the Microsoft C Run-Time Library for DOS that operate compatibly with C library routines on XENIX systems. The routines provide an identical interface to a set of operations useful on both XENIX and DOS.

The XENIX and DOS common library routines operate compatibly with UNIX library routines as well. In addition, the Microsoft C Compiler Run-Time Library for DOS contains several routines that are compatible with UNIX System V routines but that are not currently implemented on XENIX.

With the exception of error returns, the math functions in the Microsoft C Compiler Run-Time Library for DOS operate compatibly with the XENIX routines of the same names. Error returns for most math routines in the DOS library have been upgraded for compatibility with UNIX System V math-error handling.

B.2 Run-Time Routines

The sections below list routines from the DOS C library that are compatible with XENIX and UNIX System V routines. Routines specific to the DOS environment are also listed.

B.2.1 Routines Common to DOS and XENIX

The following is a list of the routines common to DOS and XENIX.

<table>
<thead>
<tr>
<th>DOS Routine</th>
<th>XENIX Routine</th>
<th>DOS Routine</th>
<th>XENIX Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort1</td>
<td>exec1p</td>
<td>getc</td>
<td>open1</td>
</tr>
<tr>
<td>abs</td>
<td>execv1</td>
<td>getc</td>
<td>perror</td>
</tr>
<tr>
<td>access1</td>
<td>execve1</td>
<td>getchar</td>
<td>pow2</td>
</tr>
<tr>
<td>acos2</td>
<td>execvp1</td>
<td>getcwd</td>
<td>printf</td>
</tr>
<tr>
<td>asctime</td>
<td>execve1</td>
<td>getenv</td>
<td>putc</td>
</tr>
<tr>
<td>asin2</td>
<td>exit</td>
<td>getpid1</td>
<td>putchar</td>
</tr>
<tr>
<td>assert</td>
<td>exp</td>
<td>gets</td>
<td>puts</td>
</tr>
<tr>
<td>atan2</td>
<td>fabs</td>
<td>getw</td>
<td>puts</td>
</tr>
<tr>
<td>atan22</td>
<td>fclose</td>
<td>gmtime</td>
<td>putw</td>
</tr>
<tr>
<td>atof</td>
<td>fclose</td>
<td>hypot</td>
<td>qsort</td>
</tr>
<tr>
<td>atoi</td>
<td>fdopen</td>
<td>isalnum</td>
<td>strdod</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isalpha</td>
<td>strdup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read1</td>
<td>strncat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strncmp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strncpy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strpbrk</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strrchr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strspn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strtod</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strtok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>strtol</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>swab</td>
</tr>
<tr>
<td>Routine</td>
<td>Routine</td>
<td>Routine</td>
<td>Routine</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>atol</td>
<td>feof</td>
<td>isascii</td>
<td>realloc</td>
</tr>
<tr>
<td>atexit$^3$</td>
<td>ferror</td>
<td>iscntrl</td>
<td>rewind</td>
</tr>
<tr>
<td>bessel$^2$,4</td>
<td>fflush</td>
<td>isdigit</td>
<td>rtmp</td>
</tr>
<tr>
<td>bsearch</td>
<td>fgetc</td>
<td>isgraph</td>
<td>sbrk</td>
</tr>
<tr>
<td>cabs</td>
<td>fgets</td>
<td>islower</td>
<td>scanf</td>
</tr>
<tr>
<td>calloc</td>
<td>fileno</td>
<td>isprint</td>
<td>setbuf</td>
</tr>
<tr>
<td>ceil</td>
<td>floor</td>
<td>ispunct</td>
<td>setjmp</td>
</tr>
<tr>
<td>chdir$^1$</td>
<td>fmod</td>
<td>isspace</td>
<td>setvbuf</td>
</tr>
<tr>
<td>chmod$^2$</td>
<td>fopen$^1$</td>
<td>issupper</td>
<td>signal$^1$</td>
</tr>
<tr>
<td>chsize</td>
<td>fprintf</td>
<td>isxdigit</td>
<td>sin$^2$</td>
</tr>
<tr>
<td>clearerr</td>
<td>fputc</td>
<td>idexp$^2$</td>
<td>sinh$^2$</td>
</tr>
<tr>
<td>close</td>
<td>fputs</td>
<td>lfind</td>
<td>sprintf</td>
</tr>
<tr>
<td>cos$^2$</td>
<td>fread$^1$</td>
<td>localtime</td>
<td>sqrt$^2$</td>
</tr>
<tr>
<td>cosh$^2$</td>
<td>free</td>
<td>locking$^1$</td>
<td>srand</td>
</tr>
<tr>
<td>creat$^1$</td>
<td>freopen$^1$</td>
<td>log$^2$</td>
<td>sscanf</td>
</tr>
<tr>
<td>ctime</td>
<td>frexp</td>
<td>log10$^2$</td>
<td>stat$^1$</td>
</tr>
<tr>
<td>difftime</td>
<td>fscanf</td>
<td>longjmp</td>
<td>strcat</td>
</tr>
<tr>
<td>dup</td>
<td>fseek$^1$</td>
<td>lsearch</td>
<td>strchr</td>
</tr>
<tr>
<td>dup2</td>
<td>fstat$^1$</td>
<td>lseek$^1$</td>
<td>strcmp</td>
</tr>
<tr>
<td>ecvt</td>
<td>ftell$^1$</td>
<td>malloc</td>
<td>strncpy</td>
</tr>
<tr>
<td>exec$^1$</td>
<td>ftime$^1$</td>
<td>mktemp</td>
<td>strcspn</td>
</tr>
<tr>
<td>execle$^1$</td>
<td>fwrite$^1$</td>
<td>modf</td>
<td>strdup</td>
</tr>
<tr>
<td>execlp$^1$</td>
<td>gcvt</td>
<td>onexit$^3$</td>
<td>strerror</td>
</tr>
</tbody>
</table>

1. Operates differently or has different meaning under DOS than under XENIX. The differences are detailed in Section B.5.

2. Implements UNIX System V-style error returns.

3. atexit (DOS) is identical to the Xenix/Unix function named onexit. See Section B.5.

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

---

**B.2.2 Routines Common to DOS and UNIX System V**

The XENIX-compatible routines listed in the previous section are also compatible with the routines of the same names in UNIX System V environments. In addition, the following DOS routines are compatible with UNIX System V routines by the same name. These routines are not implemented on XENIX.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloca</td>
<td>memchr</td>
<td>memicomp</td>
<td>memset</td>
</tr>
<tr>
<td>matherr</td>
<td>memcmp</td>
<td>memcmp</td>
<td>putenv</td>
</tr>
</tbody>
</table>
Note that most of the math functions in the DOS library implement error handling in the same manner as the UNIX System V routines of the same name. The math routines marked with a dagger (†) in the list of common routines for DOS and XENIX (see Section B.2.1) implement System V-style error handling.

### B.2.3 Routines Specific to DOS

The routines listed below are available only in the DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these routines.

```plaintext
FP_OFF
FP_SEG
bdos
cgets
__clear87
__control87
cprintf
cputs
cscanf
dieetombsbin
dmsbintomiee
dosextterr
eof
__exit
fcloseall
__ffree
fgetchar
freeetombsbin
fpilelength
flushall
__fmalloc
fmsbintomiee
__fmsize
__fpreset
fpputchar
__freetect
getch
getche
halloc
hfree
inp
int86
int86x
inttdos
inttdosx
isatty
itoa
kbhit
labs
ltoa
memavl
mkdir
movedata
__msize
__nfree
__nmalloc
__status87
rcmpi
strlwr
strncmpi
strncpy
strlen
strset
strrev
strset
strstr
strupr
sopen	
tell
spawnl
spawnle
spawnlpe
spawnv
```

### B.2.4 ANSI Extensions

The Microsoft C Run-time Library includes 134 routines that conform to the Draft Proposed American National Standard—Programming Language C. In addition, the routines listed below are extensions to the ANSI standard set of functions. Don’t use these routines in programs which must strictly adhere to the ANSI set.
<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>execle</td>
<td>getch</td>
<td>mktemp</td>
<td>stat</td>
<td></td>
</tr>
<tr>
<td>alloca</td>
<td>execv</td>
<td>getche</td>
<td>movedata</td>
<td>_status87</td>
<td></td>
</tr>
<tr>
<td>assert</td>
<td>execve</td>
<td>getcwd</td>
<td>_msize</td>
<td>strleni</td>
<td></td>
</tr>
<tr>
<td>bdos</td>
<td>execvp</td>
<td>getpid</td>
<td>_nfree</td>
<td>strdupi</td>
<td></td>
</tr>
<tr>
<td>bessel1</td>
<td>execvpe</td>
<td>getw</td>
<td>_nmalloc</td>
<td>stricmpi</td>
<td></td>
</tr>
<tr>
<td>cabs</td>
<td>_exit</td>
<td>halloc</td>
<td>_nmsize</td>
<td>strlwr</td>
<td></td>
</tr>
<tr>
<td>cgets</td>
<td>_expand</td>
<td>hfree</td>
<td>open</td>
<td>strnicmp</td>
<td></td>
</tr>
<tr>
<td>chdir</td>
<td>fcloseall</td>
<td>hypot</td>
<td>outp</td>
<td>strnset</td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td>fcvt</td>
<td>inp</td>
<td>putch</td>
<td>strrev</td>
<td></td>
</tr>
<tr>
<td>chsize</td>
<td>fdopen</td>
<td>int86</td>
<td>putenv</td>
<td>strset</td>
<td></td>
</tr>
<tr>
<td>_clear87</td>
<td>_ffree</td>
<td>int86x</td>
<td>putw</td>
<td>strupr</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>fgetchar</td>
<td>intdos</td>
<td>read</td>
<td>swab</td>
<td></td>
</tr>
<tr>
<td>control87</td>
<td>fiteetomsbin</td>
<td>intdosx</td>
<td>rmdir</td>
<td>tell</td>
<td></td>
</tr>
<tr>
<td>cprintf</td>
<td>filelength</td>
<td>isascii</td>
<td>rntemp</td>
<td>tempnam</td>
<td></td>
</tr>
<tr>
<td>cputs</td>
<td>fileno</td>
<td>isatty</td>
<td>sbrok</td>
<td>toascii</td>
<td></td>
</tr>
<tr>
<td>creat</td>
<td>flushall</td>
<td>itoa</td>
<td>segread</td>
<td>_tolower</td>
<td></td>
</tr>
<tr>
<td>cscanf</td>
<td>_fmalloc</td>
<td>kbhit</td>
<td>setmode</td>
<td>_toupper</td>
<td></td>
</tr>
<tr>
<td>dieeetomsbin</td>
<td>fmsbintoeiee</td>
<td>lfnd</td>
<td>sopen</td>
<td>tzset</td>
<td></td>
</tr>
<tr>
<td>dmsbintoeieee</td>
<td>_fmsize</td>
<td>lsearch</td>
<td>spawnl</td>
<td>ultoa</td>
<td></td>
</tr>
<tr>
<td>dosexterr</td>
<td>FP_OFF</td>
<td>locking</td>
<td>spawnln</td>
<td>umask</td>
<td></td>
</tr>
<tr>
<td>dup</td>
<td>FP_SEG</td>
<td>lseek</td>
<td>spawnlp</td>
<td>ungetch</td>
<td></td>
</tr>
<tr>
<td>dup2</td>
<td>_fpreset</td>
<td>ltoa</td>
<td>spawnlpe</td>
<td>unlink</td>
<td></td>
</tr>
<tr>
<td>ecvt</td>
<td>fputchar</td>
<td>matherr</td>
<td>spawnv</td>
<td>utime</td>
<td></td>
</tr>
<tr>
<td>eof</td>
<td>_freect</td>
<td>_memavl</td>
<td>spawnve</td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>execl</td>
<td>fstat</td>
<td>memccpy</td>
<td>spawnvnp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>execle</td>
<td>ftime</td>
<td>memicmp</td>
<td>stackavpe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exclp</td>
<td>gcvt</td>
<td>mkdir</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 The bessel routine does not correspond to a single function, but to six functions named j0, j1, jn, y0, y1, and yn.
B.3 Global Variables

The sections below list global variables used in the DOS C library that are also used in XENIX and UNIX environments. The variables specific to the DOS environment are also listed.

B.3.1 Variables Common to DOS and XENIX

The following is a list of global variables used in the run-time library and available in both the DOS and XENIX environments:

daylight
environ
errno
sys_errlist
sys_nerr
timezone
tzname

Note

Not all values of errno available on XENIX are used by the DOS run-time library.

B.3.2 Variables Common to DOS and UNIX System V

The XENIX-compatible global variables listed in the Section B.3.1 are also available in UNIX System V environments. There are no additional variables common to DOS and UNIX System V.
B.3.3 Variables Specific to DOS

The following global variables are available only in the DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these variables.

_doserrno
_osmajor
_psp
_fmode
_osminor

B.4 Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. Include files provided with the DOS C library are compatible with include files of the same name on XENIX and UNIX systems. Some additional include files are compatible with include files of the same name in UNIX System V environments.

Sections B.4.1 and B.4.2 list the DOS include files that are compatible with XENIX and UNIX System V. The include files that apply only to DOS environments are listed in Section B.4.3.

B.4.1 Include Files Common to DOS and XENIX

The following DOS include files are compatible with the XENIX (and UNIX) include files of the same name:

<table>
<thead>
<tr>
<th>assert.h</th>
<th>math.h</th>
<th>stdio.h</th>
<th>sys\timeb.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctype.h</td>
<td>setjmp.h</td>
<td>sys\locking.h</td>
<td>sys\types.h</td>
</tr>
<tr>
<td>errno.h</td>
<td>signal.h</td>
<td>sys\stat.h</td>
<td>time.h</td>
</tr>
<tr>
<td>fcntl.h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.4.2 Include Files Common to DOS and UNIX System V

The XENIX-compatible include files listed in Section B.4.1 are also compatible with the include files of the same names in UNIX System V environments. In addition, the names of the following DOS include files correspond to UNIX System V include files; however, the DOS include files may not contain all the constants and types defined in the corresponding UNIX System V include files.

- malloc.h
- string.h
- memory.h
- varargs.h
- search.h

B.4.3 Include Files Specific to DOS

The following include files are used only in DOS environments and do not have counterparts on XENIX and UNIX systems:

- conio.h
- direct.h
- dos.h
- io.h
- process.h
- share.h
- stderr.h
- stdlib.h

B.4.4 ANSI Include Files

The following table lists the include files necessary to use the ANSI runtime library.

- assert.h
- math.h
- stdio.h
- ctype.h
- setjmp.h
- stdlib.h
- float.h
- signal.h
- string.h
- limits.h
- stdarg.h
- time.h
B.5 Differences Between Routines Common to DOS and XENIX

Sections B.5.1 through B.5.25 explain how the DOS routines in the common library for XENIX and DOS differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of DOS functions provided in the reference section (Part 2 of this manual) and with the descriptions of the XENIX routines in the appropriate XENIX manual.

B.5.1 abort

The DOS version of the `abort` routine terminates the process by a call to an exit routine rather than through a signal. Control is returned to the parent (calling) process with an exit status of 3 and the following message is printed to standard error:

Abnormal program termination

No core dump occurs on DOS.

B.5.2 access

The `access` routine checks the access to a given file. Under DOS, the real and effective user IDs are nonexistent. The permission (access) setting can be any combination of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>Read</td>
</tr>
<tr>
<td>02</td>
<td>Write</td>
</tr>
<tr>
<td>00</td>
<td>Check for existence</td>
</tr>
</tbody>
</table>

The "Execute" access mode (01) is not implemented.

In case of error, only the `EACCES` and `ENOENT` values may be returned for `errno` on DOS.
B.5.3 atexit

The DOS function `atexit` is identical to the XENIX/UNIX function `onexit`, save for the name and the type declaration. The `atexit` function is declared as `int`, while `onexit` is declared as `onexit_t`.

B.5.4 chdir

In case of error, only the `ENOENT` value may be returned for `errno` on DOS.

B.5.5 chmod

The `chmod` routine can set the “owner” access permissions for a given file, but all other permission settings are ignored. The mode argument can be any one of the constant expressions shown in the left-most column below; the equivalent XENIX value is shown in the right-most column:

<table>
<thead>
<tr>
<th>Constant Expression</th>
<th>Meaning</th>
<th>XENIX Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S_IREAD</code></td>
<td>Read by owner</td>
<td>0400</td>
</tr>
<tr>
<td><code>S_IWRITE</code></td>
<td>Write by owner</td>
<td>0200</td>
</tr>
<tr>
<td>`S_IREAD</td>
<td>S_IWRITE`</td>
<td>Read and write by owner</td>
</tr>
</tbody>
</table>

The `S_IREAD` and `S_IWRITE` constants are defined in the `sys\stat.h` include file. Note that the OR operator (`|`) is used to combine these constants to form read and write permission.

If write permission is not given, the file is treated as a read-only file. Giving write-only permission is allowed, but has no effect; under DOS, all files are readable.

In case of error, only the `ENOENT` value may be returned for `errno` on DOS.
B.5.6 creat

The creat routine creates a new file or prepares an existing file for writing. If the file is created, the access permissions are set as defined by the mode argument. Only “owner” permissions are allowed (see chmod above).

In case of error, only the EACCES, EMFILE, and ENOENT values may be returned for errno on DOS.

Use of the open routine is preferred over creat when creating or opening files in both DOS and XENIX environments.

B.5.7 exec

The DOS versions of the execl, execle, execlp, execlpe, execv, execve, execvpe, and execvp routines overlay the calling process, as in the XENIX environment. If there is not enough memory for the new process, the exec routine will fail and return to the calling process. Otherwise, the new process begins execution.

Under DOS, the exec routines do not perform the following functions:

- Use the close-on-exec flag to determine open files for the new process.
- Disable profiling for the new process (profiling is not available under DOS).
- Pass signal settings to the child process. Under DOS, all signals (including signals set to be ignored) are reset to the default in the child process.

The combined size of all arguments (including the program name) in an exec routine under DOS must not exceed 128 bytes.

In case of error, the E2BIG, EACCES, ENOENT, ENOEXEC, and ENOMEM values may be returned for errno on DOS. In addition, the EMFILE value may be used; under DOS, the file must be opened to determine whether or not it is executable.
B.5.8 fopen, freopen

The DOS versions of the fopen and freopen routines open stream files just as they do in the XENIX environment. However, under DOS the following additional values for the type string are available:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Opens the file in text mode. Opening a file in this mode causes translation of carriage-return-line-feed (CR-LF) character combinations into a single line feed (LF) on input. Similarly, on output, line feeds are translated into CR-LF combinations.</td>
</tr>
<tr>
<td>b</td>
<td>Opens the file in binary mode. This mode suppresses translation.</td>
</tr>
</tbody>
</table>

See the DOS reference pages (in Part 2 of this manual) for the fopen and freopen routines to obtain more information on the default mode setting.

The DOS and XENIX versions of these routines also differ in their interpretation of append mode (a or a+). When append mode is specified in the DOS version of fopen or freopen, the file pointer is repositioned at the end of the file before any write operation. Thus all write operations take place at the end of the file.

In the XENIX versions, all write operations take place at the current position of the file pointer. In append mode, the file pointer is initially positioned at the end of the file, but if the file pointer is later repositioned, write operations take place at the new position rather than at the end of the file.

B.5.9 fread

The DOS fread routine uses the low-level read function to carry out read operations. If the file has been opened in text mode, read replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting the end of the file.
B.5.10 fseek

The DOS version of the fseek routine moves the file pointer to the given position, just as in the XENIX environment. However, for streams opened in text mode, fseek has limited use because carriage-return-line-feed translations can cause fseek to produce unexpected results. The only fseek operations guaranteed to work on streams opened in text mode are: seeking with an offset of 0 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.

B.5.11 fstat

DOS does not make as much information available for file handles as it does for full path names; thus the DOS version of fstat returns less useful information than does the stat routine. The DOS fstat routine can detect device files, but it must not be used with directories.

The structure returned by fstat contains the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_mode</td>
<td>User read and write bits reflect the file’s permission setting. The S_IFCHR bit is set for a device; otherwise, the S_IFREG bit is set.</td>
</tr>
<tr>
<td>st_ino</td>
<td>Not used.</td>
</tr>
<tr>
<td>st_dev</td>
<td>Either the drive number of the disk containing the file, or the file handle in the case of a device (same as st_rdev).</td>
</tr>
<tr>
<td>st_rdev</td>
<td>Either the drive number of the disk containing the file, or the file handle in the case of a device (same as st_dev).</td>
</tr>
<tr>
<td>st_nlink</td>
<td>Always 1.</td>
</tr>
<tr>
<td>st_uid</td>
<td>Not used.</td>
</tr>
<tr>
<td>st_gid</td>
<td>Not used.</td>
</tr>
<tr>
<td>st_size</td>
<td>Size, in bytes, of the file.</td>
</tr>
<tr>
<td>st_atime</td>
<td>Time of last modification of file (same as st_mtime and st_ctime).</td>
</tr>
</tbody>
</table>
**st_mtime**  
Time of last modification of file (same as **st_atime** and **st_ctime**).

**st_ctime**  
Time of last modification of file (same as **st_atime** and **st_mtime**).

In case of error, only the **EBADF** value may be returned for **errno** on DOS.

### B.5.12 ftell

The DOS version of the **ftell** routine gets the current file pointer position, just as in the XENIX environment. However, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, since text mode causes carriage-return-line-feed translation. The **ftell** routine can be used in conjunction with the **fseek** routine to remember and return to file locations correctly.

### B.5.13 ftime

Unlike the system time on XENIX systems, the DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of an DOS environment variable named **TZ** to determine the time zone. The user can set the default time zone by setting the **TZ** variable. If **TZ** is not explicitly set, the default time zone corresponds to the Pacific time zone. See the reference page for **tzset** in Part 2 of this manual for details on the **TZ** variable.

### B.5.14 fwrite

The DOS **fwrite** routine uses the low-level **write** function to carry out write operations. If the file was opened in text mode, every line-feed (**LF**) character in the output is replaced by a carriage-return—line-feed (**CR-LF**) pair before being written. This does not affect the return value.

### B.5.15 getpid

The **getpid** routine returns a process-unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process identification returned by **getpid** in the XENIX environment.
B.5.16 locking

The DOS and XENIX versions of the locking routine differ in several respects, as listed below:

1. Under DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. This means that setting LK_RLCK in the locking call is equivalent to setting LK_LOCK, and setting LK_NBRLCK is equivalent to setting LK_NBLCK.

2. On DOS, specifying LK_LOCK or LK_RLCK will not cause a program to wait until the specified region of a file is unlocked. Instead, up to ten attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the locking function returns an error value.

   On XENIX, if the first attempt at locking fails, the locking process "sleeps" (suspends execution) and periodically "wakes" to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.

3. On DOS, locking of overlapping regions of a file is not allowed.

4. On DOS, if more than one region of a file is locked, only one region can be unlocked at a time, and the region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

B.5.17 lseek

In case of error, only the EBADF and EINVAL values may be returned for errno on DOS.

B.5.18 open

The open routine opens a file handle for a named file, just as in the XENIX environment. However, two additional oflag values (O_BINARY and O_TEXT) are available and the O_NDELAY and O_SYNCW values are not available.
The **O_BINARY** flag causes the file to be opened in binary mode, regardless of the default mode setting. Similarly, the **O_TEXT** flag causes the file to be opened in text mode.

In case of error, only the **EACCES**, **EEXIST**, **EMFILE**, and **ENOENT** values may be used for **errno** on DOS.

### B.5.19 read

The DOS version of the **read** routine reads characters from the file given by a file handle, just as in the XENIX environment. However, if the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting an end-of-file condition.

In case of error, only the **EBADF** value may be used for **errno** on DOS.

### B.5.20 signal

The DOS version of the **signal** routine can only handle the **SIGINT** and **SIGFPE** signals. In DOS, **SIGINT** is defined to be INT 23H (the signal), while **SIGFPE** corresponds to floating-point exceptions that are not masked.

On DOS, child processes executed through the **exec** or **spawn** routines do not inherit the signal settings of the parent process. All signal settings (including signals set to be ignored) are reset to the default settings in the child process.

The DOS version of **signal** uses only **EINVAL** for **errno**.

### B.5.21 stat

The **stat** routine returns a structure defining the current status of the given file or directory. The structure members returned by **stat** have the following names and meanings on DOS:

#### Value
Meaning

`st_mode`  User read and write bits reflect the file's permission setting. The `S_IFDIR` bit is set for a device; otherwise, the `S_IFREG` bit is set.

`st_ino`  Not used.

`st_dev`  Drive number of the disk containing the file (same as `st_rdev`).

`st_rdev`  Drive number of the disk containing the file (same as `st_dev`).

`st_nlink`  Always 1.

`st_uid`  Not used.

`st_gid`  Not used.

`st_size`  Size, in bytes, of the file.

`st_atime`  Time of last modification of file (same as `st_mtime` and `st_ctime`).

`st_mtime`  Time of last modification of file (same as `st_atime` and `st_atime` and `st_ctime`).

`st_ctime`  Time of last modification of file (same as `st_atime` and `st_mtime`).

In case of error, only the `ENOENT` value may be returned for `errno` on DOS.

### B.5.22 system

The `system` routine passes the given string to the operating system for execution. For DOS to execute this string, the full path name of the directory containing it must be assigned to the environment variable. The `system` call returns an error if the string cannot be found using these variables.

In case of error, only the `E2BIG`, `ENOENT`, `ENOEXEC`, and `ENOMEM` values may be returned for `errno` on DOS.
B.5.23 umask

The **umask** routine can set a mask for “owner” read and write access permissions only. All other permissions are ignored. (See the discussion of the **access** routine above for details.)

B.5.24 unlink

The DOS version of the **unlink** routine always deletes the given file. Since DOS does not implement multiple “links” to the same file, unlinking a file is the same as deleting it.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on DOS.

B.5.25 utime

The DOS **utime** routine sets the file modification time only; DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** on DOS. In addition, the **EMFILE** value may be used; under DOS, the file must be opened to set the modification time.

B.5.26 write

The **write** routine writes a specified number of characters to the file named by the given file handle, just as in the XENIX environment. However, if the file has been opened in text mode, every line-feed (LF) character in the output is replaced by a carriage-return-line-feed (CR-LF) pair before being written. This does not affect the return value.

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on DOS.
<table>
<thead>
<tr>
<th>Reference Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ (forward slash), as path-name delimiter, 20</td>
</tr>
<tr>
<td>abort XENIX version, differences from, 550, 66, 94</td>
</tr>
<tr>
<td>abs, 73</td>
</tr>
<tr>
<td>abs function, 96</td>
</tr>
<tr>
<td>abs macro, 92</td>
</tr>
<tr>
<td>Absolute value</td>
</tr>
<tr>
<td>abs function, 96</td>
</tr>
<tr>
<td>cabs function, 123</td>
</tr>
<tr>
<td>fabs function, 182</td>
</tr>
<tr>
<td>labs function, 302</td>
</tr>
<tr>
<td>access XENIX version, differences from, 550, 45, 97</td>
</tr>
<tr>
<td>Access mode, 187, 218, 237</td>
</tr>
<tr>
<td>acos floating-point support, 25, 58, 99</td>
</tr>
<tr>
<td>alloca, 60, 101</td>
</tr>
<tr>
<td>Allocation. See Memory allocation ambksz variable, 31</td>
</tr>
<tr>
<td>ANSI</td>
</tr>
<tr>
<td>include files, 549</td>
</tr>
<tr>
<td>run-time extensions, 545</td>
</tr>
<tr>
<td>Appending, 188, 218, 237, 367</td>
</tr>
<tr>
<td>Arc</td>
</tr>
<tr>
<td>cosine function, 99</td>
</tr>
<tr>
<td>sine function, 105</td>
</tr>
<tr>
<td>tangent function, 109</td>
</tr>
<tr>
<td>arg0, DOS considerations, 24</td>
</tr>
<tr>
<td>Argument lists, variable, 524, 529</td>
</tr>
<tr>
<td>Argument singularity, 328</td>
</tr>
<tr>
<td>Argument type checking, 4, 16, 17</td>
</tr>
<tr>
<td>Argument type lists, 16</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
<tr>
<td>macros, with side effects, 13</td>
</tr>
<tr>
<td>notational conventions, 6</td>
</tr>
<tr>
<td>variable number, 17, 73</td>
</tr>
<tr>
<td>variable-length number, 524, 529</td>
</tr>
<tr>
<td>asctime, 72, 103</td>
</tr>
<tr>
<td>asin floating-point support, 25, 58, 105</td>
</tr>
<tr>
<td>assert, 73, 107</td>
</tr>
<tr>
<td>assert.h, 74</td>
</tr>
<tr>
<td>assert.h, contents, 80</td>
</tr>
<tr>
<td>Assertions, 107</td>
</tr>
<tr>
<td>atan floating-point support, 25, 58, 109</td>
</tr>
<tr>
<td>atan2 floating-point support, 25, 58, 109</td>
</tr>
<tr>
<td>atexit XENIX version, differences from, 551, 66, 111</td>
</tr>
<tr>
<td>atof floating-point support, 25, 44, 113</td>
</tr>
<tr>
<td>atoi, 44, 113</td>
</tr>
<tr>
<td>atol, 44, 113</td>
</tr>
<tr>
<td>Backslash () as escape character, 21 path-name delimiter, 20</td>
</tr>
<tr>
<td>bdos, 65, 116</td>
</tr>
<tr>
<td>bessel floating-point support, 25, 58, 118</td>
</tr>
<tr>
<td>Bessel functions, 118</td>
</tr>
<tr>
<td>Binary</td>
</tr>
<tr>
<td>format, conversion to IEEE double precision, 157</td>
</tr>
<tr>
<td>floating point, 207</td>
</tr>
<tr>
<td>int reading, 271</td>
</tr>
<tr>
<td>writing, 391</td>
</tr>
<tr>
<td>mode, 22, 34, 188, 219, 237, 238, 367, 428</td>
</tr>
<tr>
<td>search, 120, 306</td>
</tr>
<tr>
<td>BINMODE.OBJ, 22, 34</td>
</tr>
<tr>
<td>Bold capitals, use of, 6</td>
</tr>
<tr>
<td>Bold font, use of, 6</td>
</tr>
<tr>
<td>Break value, 413</td>
</tr>
<tr>
<td>bsearch, 70, 120</td>
</tr>
<tr>
<td>Buffer manipulation include file, 42</td>
</tr>
<tr>
<td>memccpy, 41, 333</td>
</tr>
<tr>
<td>memchr, 41, 335</td>
</tr>
<tr>
<td>memcmp, 41, 337</td>
</tr>
<tr>
<td>memcpy, 41, 339</td>
</tr>
<tr>
<td>memcmp, 41, 341</td>
</tr>
<tr>
<td>memmove, 41, 344</td>
</tr>
</tbody>
</table>
### Reference Index

<table>
<thead>
<tr>
<th>Category</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Buffer manipulation</strong></td>
<td>41, 345</td>
</tr>
<tr>
<td><code>memset</code></td>
<td></td>
</tr>
<tr>
<td>movedata</td>
<td>41, 354</td>
</tr>
<tr>
<td><strong>Buffering</strong></td>
<td>52, 47, 52</td>
</tr>
<tr>
<td>preopened streams</td>
<td></td>
</tr>
<tr>
<td><strong>Buffers</strong></td>
<td>423</td>
</tr>
<tr>
<td>assigning</td>
<td></td>
</tr>
<tr>
<td>comparing</td>
<td>337, 341</td>
</tr>
<tr>
<td>copying</td>
<td></td>
</tr>
<tr>
<td>overlapping moves</td>
<td>339, 333, 339, 354</td>
</tr>
<tr>
<td>flushing</td>
<td>52, 194, 212</td>
</tr>
<tr>
<td>searching</td>
<td>335</td>
</tr>
<tr>
<td>setting characters in</td>
<td>345</td>
</tr>
<tr>
<td>BUFSIZ constant</td>
<td>50, 88</td>
</tr>
<tr>
<td><strong>Byte order, swapping</strong></td>
<td>492</td>
</tr>
<tr>
<td>BYTEREGS type</td>
<td>82</td>
</tr>
<tr>
<td><strong>cabs</strong></td>
<td>25, 58, 123</td>
</tr>
<tr>
<td>floating-point support</td>
<td></td>
</tr>
<tr>
<td><strong>calloc</strong></td>
<td>60, 125</td>
</tr>
<tr>
<td><strong>Carriage-return–line-feed translation</strong></td>
<td>See Binary mode or Text mode</td>
</tr>
<tr>
<td><strong>Carry flag</strong></td>
<td>116, 283, 285, 288, 290</td>
</tr>
<tr>
<td><strong>Case sensitivity</strong></td>
<td></td>
</tr>
<tr>
<td>C language</td>
<td>19</td>
</tr>
<tr>
<td>DOS</td>
<td>19</td>
</tr>
<tr>
<td>XENIX</td>
<td>19</td>
</tr>
<tr>
<td><strong>ceil</strong></td>
<td>25, 127</td>
</tr>
<tr>
<td>floating-point support</td>
<td></td>
</tr>
<tr>
<td>Ceiling function</td>
<td>127</td>
</tr>
<tr>
<td><code>;C_FILE_INFO</code></td>
<td>445</td>
</tr>
<tr>
<td><code>cgets</code></td>
<td>57, 128</td>
</tr>
<tr>
<td><strong>Character</strong></td>
<td></td>
</tr>
<tr>
<td>classification and conversion</td>
<td>42</td>
</tr>
<tr>
<td><code>tolower</code></td>
<td>507</td>
</tr>
<tr>
<td><code>toupper</code></td>
<td>507</td>
</tr>
<tr>
<td><strong>Characters</strong></td>
<td></td>
</tr>
<tr>
<td>conversion to ASCII</td>
<td>507</td>
</tr>
<tr>
<td>conversion to lowercase</td>
<td>507</td>
</tr>
<tr>
<td>conversion to uppercase</td>
<td>507</td>
</tr>
<tr>
<td>reading</td>
<td>198, 260, 398</td>
</tr>
<tr>
<td>reading from console with echo</td>
<td>264, 262</td>
</tr>
<tr>
<td>reading from port</td>
<td>282</td>
</tr>
<tr>
<td>ungetting</td>
<td>516, 518</td>
</tr>
<tr>
<td>writing</td>
<td>228, 384, 532</td>
</tr>
<tr>
<td>writing to console</td>
<td>386</td>
</tr>
<tr>
<td>writing to port</td>
<td>371</td>
</tr>
<tr>
<td><strong>chdir</strong></td>
<td></td>
</tr>
<tr>
<td>XENIX version, differences from</td>
<td>551, 45, 130</td>
</tr>
<tr>
<td><strong>Child process</strong></td>
<td></td>
</tr>
<tr>
<td>effects on floating-point state of parent</td>
<td>223</td>
</tr>
<tr>
<td>signal settings</td>
<td>174, 443</td>
</tr>
<tr>
<td>translation mode</td>
<td>173, 443, 171, 443</td>
</tr>
<tr>
<td><strong>chmod</strong></td>
<td></td>
</tr>
<tr>
<td>XENIX version, differences from</td>
<td>551, 45, 132</td>
</tr>
<tr>
<td><code>clear87</code></td>
<td>58, 136</td>
</tr>
<tr>
<td><code>clearerr</code></td>
<td>19, 48, 138</td>
</tr>
<tr>
<td>Clearing end-of-file, streams</td>
<td>138</td>
</tr>
<tr>
<td>Clearing errors</td>
<td>138</td>
</tr>
<tr>
<td><strong>clock</strong></td>
<td>72, 139</td>
</tr>
<tr>
<td><code>close</code></td>
<td>54, 140</td>
</tr>
<tr>
<td><strong>Common library</strong></td>
<td></td>
</tr>
<tr>
<td>common routines listed</td>
<td>543, 544</td>
</tr>
<tr>
<td>global variables</td>
<td>547</td>
</tr>
<tr>
<td>include files</td>
<td>548</td>
</tr>
<tr>
<td>run-time routine, differences</td>
<td>550, 543</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td></td>
</tr>
<tr>
<td>differences listed</td>
<td>550</td>
</tr>
<tr>
<td>global variables</td>
<td>547</td>
</tr>
</tbody>
</table>
Compatibility (continued)
include files, 548
math routines, 543
run-time routines, 543
with UNIX, 543
with XENIX, 543
complex type, 36, 85
conio.h, 57
conio.h, contents, 80
Console, ungetting characters from, 518
control87, 58, 142
Conventions, notational, 6
Conversion
characters to ASCII, 507
characters to lowercase, 507
characters to uppercase, 507
floating-point numbers to integers and fractions, 352
floating-point numbers to strings, 167, 185, 258
IEEE double to MS binary double, 157
IEEE floating point to MS binary floating point, 207
integers to strings, 298
long integers to strings, 323, 512
MS binary double to IEEE double, 157
MS binary floating point to IEEE floating point, 207
strings to floating-point values, 113
strings to lowercase, 470
strings to uppercase, 491
cos
floating-point support, 25, 58, 144
cosh
floating-point support, 25, 58, 144
Cosine, 144
cprintf
argument-type-checking limitations, 17, 57, 146
cprintf. See also printf family
cputs, 57, 148
creat
XENIX version, differences from, 552, 54, 149
CR-LF translation. See Binary mode or Text mode
cscanf
argument-type-checking limitations, 17,
ctype.h, 43
ctype.h, contents, 80
Data conversion
atof, 44, 113
atoi, 44, 113
atol, 44, 113
ecvt, 44, 167
fcvt, 44, 185
gcvt, 44, 258
include files, 44
itoa, 44, 298
ltoa, 44, 323, 512
strstr, 44, 485
strtol, 44, 485
strtof, 44, 485
ultoa, 44
Data conversion. See also Conversions
Data items
reading, 231
writing, 255
Data type limits, 84
Date routines. See Time routines
daylight variable, 32, 510
Deallocating memory, 196, 233, 278, 359
Declarations, function. See Function declarations
Default translation mode
changing, 22
child process, used in, 173, 443
overriding, 23, 22, 34, 367, 428
Delimiters for path-name components. See Path names
dieetomsbin, 157
difftime, 72, 158
direct.h, 45
direct.h, contents, 81
Directories
changing, 130
creating, 347
current working directory, getting, 265
deleting, 408
renaming, 404
Reference Index

Directory control
  chdir, 45, 130
  chmod, 132
  getcwd, 45, 265
  include files, 45
  mkdir, 45, 347
  remove, 402
  rmdir, 45
  unlink, 520

Directory names, notational conventions, 6
  div, 73, 160

Division
  div, 160
  ldiv, 304
  div_t type, 36
  dmsbintoieee, 157

DOMAIN, 328, 540

DOS
  commands, execution of from within programs, 494
  considerations
    DOS error codes, 33
    version number, detection of, 34, 23
  interface routines
    bdos, 65, 116
    dosextterr, 65, 162
    include files, 65
    int86, 65, 283
    int86x, 65, 285
    intdos, 65, 288
    intdosx, 65, 290
    segread, 421
  interrupts
    invoking, 283, 285
    SIGINT, 432
  system calls
    error handling, 162
    invoking, 116, 288, 290
    version number, 34
  _doserrno variable, 33
  DOSERROR type, 36, 82, 162
dosextterr
  DOS considerations, 23, 65, 162
dos.h, 65
dos.h, contents, 81

Dynamic allocation. See Memory allocation

\ (backslash) as
  escape character, 21
  path-name delimiter, 20

dup2 (continued)
  DOS considerations, 23, 54, 164

dup2
  DOS considerations, 23, 54, 164

E2BIG, 538
EACCES, 538
EBADF, 538
Echoing characters, 264
ecvt, 44, 167
EDADLOCK, 538
EDOM, 538
EEXIST, 538
EINVAL, 539
Ellipses dots, use of, 7
EMFILE, 539
End-of-file
  low-level I/O, 169
  stream I/O
    clearing, 138, 406, 190
End-of-file condition, 19
ENOENT, 539
ENOEXEC, 539
ENOMEM, 539
ENOSPC, 539

environ variable, 35, 267, 387
Environment table, 35, 74, 267, 387
Environment variable names, notational conventions, 6
Environment variables, 267, 387
eof, 19, 54, 169
EOF constant, 50, 88
ERANGE, 539
errno variable, 18, 19, 33, 56, 75, 82, 87, 372, 467, 537
erro.h, 537
errno.h, contents, 82

Error handling
  DOS error codes, 33
  DOS system calls, 162
logic errors, 107
perror, 372
stream operations, 19
strerror, 467, 18

indicator, 19, 53, 138, 192

messages
messages (continued)
user supplied, 372, 467, 537
returns, 18
Euclidean distance, 280
exception type, 36, 85, 328
EXDEV, 539
exec family
DOS considerations, 24
exec routines, differences between, 68
path-name delimiters, 21
XENIX version, differences from,
552, 66, 171
exec
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
excl. See also exec family
exce
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
execl. See also exec family
exec
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
exce.
See also exec family
execl
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
execle. See also exec family
execle
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
Executing programs from within
programs, 171, 443
execv
XENIX version, differences from,
552, 66, 171
execv. See also exec family
execv
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
execve. See also exec family
execv
argument-type-checking limitations,
17
XENIX version, differences from,
552, 66, 171
execve
argument-type-checking limitations,
17, 171
FILE
pointer, 47, 50
structure, 50
type, 36, 89
File, 294
File handles
duplication, 164
predefined, 55
stream, used with, 210, 55
File handling
access, 45, 97
chmod, 45
chsize, 45, 134
filelength, 45, 208
fstat, 45, 248
include files, 46
exit, 66
exit, 176
Exiting processes, 176
exp
floating-point support, 25, 58, 178
exp
expand, 60, 179
Exponential functions
exp, 178
frexp, 240
ldexp, 303
log, 316
log10, 316
pow, 374
sqrt, 451
fabs
floating-point support, 25, 58, 182
Far pointers, 221
close, 48, 183
closeall, 48, 183
cntl.h, contents, 83
cvt, 44, 185
dopen, 48, 187
eof, 19, 48, 190
terror, 19, 48, 192
flush, 48, 194
ffree, 60, 196
fgetc, 48, 198
fgetchar, 48, 198
fgetpos, 48, 200
fget, 48, 202
fheapchk, 60, 204
fheapset, 60, 205
fheapwalk, 60, 206
fseekosbin, 207
FILE
pointer, 47, 50
structure, 50
type, 36, 89
File, 294
File handles
duplication, 164
predefined, 55
stream, used with, 210, 55
File handling
access, 45, 97
chmod, 45
chsize, 45, 134
filelength, 45, 208
fstat, 45, 248
include files, 46
exit, 66
exit, 176
Exiting processes, 176
exp
floating-point support, 25, 58, 178
exp
expand, 60, 179
Exponential functions
exp, 178
frexp, 240
ldexp, 303
log, 316
log10, 316
pow, 374
sqrt, 451
fabs
floating-point support, 25, 58, 182
Far pointers, 221
close, 48, 183
closeall, 48, 183
cntl.h, contents, 83
cvt, 44, 185
dopen, 48, 187
eof, 19, 48, 190
terror, 19, 48, 192
flush, 48, 194
ffree, 60, 196
fgetc, 48, 198
fgetchar, 48, 198
fgetpos, 48, 200
fget, 48, 202
fheapchk, 60, 204
fheapset, 60, 205
fheapwalk, 60, 206
fseekosbin, 207
FILE
pointer, 47, 50
structure, 50
type, 36, 89
File, 294
File handles
duplication, 164
predefined, 55
stream, used with, 210, 55
File handling
access, 45, 97
chmod, 45
chsize, 45, 134
filelength, 45, 208
fstat, 45, 248
include files, 46
exit, 66
exit, 176
Exiting processes, 176
exp
floating-point support, 25, 58, 178
exp
expand, 60, 179
Exponential functions
exp, 178
frexp, 240
ldexp, 303
log, 316
log10, 316
pow, 374
sqrt, 451
fabs
floating-point support, 25, 58, 182
Far pointers, 221
close, 48, 183
closeall, 48, 183
cntl.h, contents, 83
cvt, 44, 185
dopen, 48, 187
eof, 19, 48, 190
terror, 19, 48, 192
flush, 48, 194
ffree, 60, 196
fgetc, 48, 198
fgetchar, 48, 198
fgetpos, 48, 200
fget, 48, 202
fheapchk, 60, 204
fheapset, 60, 205
fheapwalk, 60, 206
fseekosbin, 207
FILE
pointer, 47, 50
structure, 50
type, 36, 89
File, 294
File handles
duplication, 164
predefined, 55
stream, used with, 210, 55
File handling
access, 45, 97
chmod, 45
chsize, 45, 134
filelength, 45, 208
fstat, 45, 248
include files, 46
File handling (continued)
isatty, 45
locking, 45, 311
mktemp, 45, 349
remove, 45
rename, 45, 404
setmode, 45, 428
stat, 45, 458
umask, 45, 514
unlink, 45
File names, notational conventions, 6
File permission mask. See Permission mask
File pointer, 53, 56
File pointer positioning, 200, 243, 246,
251, 320, 406, 499
File status information, 248, 458
filelength, 45, 208
File-name conventions, 19
fileno, 48, 55, 210
Files
changing size of, 134
closing, 57, 140
creating, 149, 367
deleting, 402, 520
file pointer, positioning, 320, 499
length, determination of, 208
locking, 311
modification time, setting of, 522
opening, 54, 149, 367
reading characters from, 398
renaming, 404
status information, 248, 458
temporary, 349
writing characters to, 532
float.h, contents, 83
Floating point not loaded, 26
Floating-point
control word, getting and setting, 142
events, recovery from, 223
exceptions, 83, 432
math package
  _ clear87, 136
  _ control87, 142
  _ fpreset, 223
  _ reinitialization, 223
  _ status87, 461
numbers, conversion to strings, 167,
  185, 258
ranges, 83
Floating-point (continued)
status word, 136, 461
support, 25
floor
  floating-point support, 25, 58, 211
flushall, 48, 212
Flushing buffers, 52, 194, 212
_fmalloc, 60, 214
fmod
  floating-point support, 25, 58, 216
  _ fmode variable, 22, 34
fmsbintoieee, 207
  _ fsize, 60, 217
fopen
  default translation mode, changing, 22
  default translation mode, overriding, 23
  XENIX version, differences from, 553, 48, 218
Formatted I/O, 146, 152, 226, 241, 376,
415, 449, 454, 529
Forward slash (/), as path-name delimiter, 20
FP_ OFF, 66, 221
  _ fpreset, 58, 223
fprintf
  argument-type-checking limitations, 17, 48, 226
fprintf. See also printf family
FP_ SEG, 66, 221
fputc, 48, 228
fputchar, 48, 228
fputs, 48, 230, 390
fread
  XENIX version, differences from, 553, 48, 231
free, 60, 233
  _ freect, 60, 235
Freeing memory blocks, 196, 233, 278,
  359
freopen
  XENIX version, differences from, 553, 48, 237
frexp
  floating-point support, 25, 58, 240
fscanf
  argument-type-checking limitations, 17, 48, 241
fscanf. See also scanf family
fseek
fseek (continued)
  XENIX version, differences from, 554, 48, 243
fsetpos, 48, 246
fstat
  XENIX version, differences from, 554, 48, 248
ftell
  XENIX version, differences from, 555, 48, 251
ftime
  XENIX version, differences from, 555, 72, 253
Function declarations, 16, 79
Functions, advantages over macros, 12
fwrite
  XENIX version, differences from, 555, 48, 255
Appending, 438
Binary mode, 438
Compatibility mode, 438
Default, 438
Files, creating, 438
Files, opening, 438
I/O, low-level sopen, 438
oflag. See Open
Open, 438
Permission, 438
sopen, 438
Text, 438
gcvt, 44, 258
getc, 48, 260
getch, 57, 262
getchar, 48, 260
getch, 57, 264
getcwd, 45, 265
getenv, 73, 267
getpid
  XENIX version, differences from, 555, 66, 269
gets, 48, 270
getw, 48, 271
gmtime, 72, 273
Goto, nonlocal, 75, 318, 425
Greenwich mean time, 72, 273
haloc, 60, 276
Handle. See File handle
hfree, 60, 278
HUGE, 85
Huge arrays, use in library functions, 26
Huge pointers, use in library functions, 26
HUGE._ VAL, 85
HUGE._ VAL. See also HUGE
Hyperbolic cosine, 144
Hyperbolic sine, 436
Hyperbolic tangent, 497
hypot
  floating-point support, 25, 58, 280
Hypotenuse, 280
Identifiers, notational conventions, 6
IEEE, converting to Microsoft binary
  double precision, 157
  floating point, 207
Include files
  assert.h, 80
  buffer manipulation routines, used
    with, 42
  character classification and
    conversion, 43
  common library, use in, 548
  conio.h, 80
  console and port I/O, 57
cctype.h, 80
data conversion, 44
direct.h, 81
directory control, 45
DOS interface routines, 65
dos.h, 81
errno.h, 82
cfcntl.h, 83
file handling, 46
float.h, 83
function declarations, 79
io.h, 83
limits.h, 84
low-level I/O, 54
malloc.h, 84
math routines, 60
math.h, 84
memory allocation, 62
memory.h, 85
miscellaneous routines, 74
naming conventions, 4
notational conventions, 6
Reference Index

Include files (continued)
process control, 68
process.h, 85
processor calls, 66
search.h, 86
searching and sorting, 70
setjmp.h, 86
share.h, 86
signal.h, 87
stdarg.h, 87
stddef.h, 87
stdio.h, 88
stdlib.h, 89
stream I/O, 50
string manipulation, 71
string.h, 90
sys\locking.h, 90
sys\stat.h, 90
time.h, 91
sys\types.h, 91
sys\utime.h, 91
time routines, 72, 73
time.h, 91
v2tov3.h, 92
varargs.h, 92

Initializing buffers. See Setting
characters
inp, 57, 282
Input and output. See I/O
Input/Output. See I/O
int86, 65, 283
int86x, 65, 285
intdos, 65, 288
intdosx, 65, 290
Integers
long, conversion to strings, 323, 512
Integers, conversion to strings, 298
Interrupt signals, 432
Interrupts. See DOS interrupts
I/O
buffered, 47
console and port (continued)
outp, 57, 371
putch, 57, 386
ungeth, 57, 518, 47
low level
use of errno, 19, 47
low-level
close, 54, 140
creat, 54, 149
dup, 54, 164
dup2, 54, 164
eof, 54, 169
error handling, 19, 56
include files, 54
lseek, 54, 320
open, 54, 367
read, 54, 398
sopen, 54
tell, 54, 499
write, 54, 532, 47
_stream, 47, 48
__iob array, 89
io.h, 46, 54
io.h, contents, 83
isalnum, 42, 292
isalpha, 292
isascii, 292
isatty, 45, 294
iscntrl, 295
isdigit, 42, 295
isgraph, 42, 295
islower, 42, 295
isprint, 42, 295
ispunct, 42, 295
isspace, 42, 295
isupper, 42, 295
isxdigit, 42, 295
Italics, use of, 6
itoa, 44, 298, 512

j0. See Bessel
j1. See Bessel
jmp_ buf type, 36
jn. See Bessel

kbhit, 57, 301
Keys, key sequences, notational
conventions, 8
Keystroke, testing for, 301
Keywords, notational conventions, 6

labs, 73, 302
ldexp
  floating-point support, 25, 58, 303
ldiv, 73, 304
ldiv_t type, 36
Length
  of files, 208
  of strings, 469
lfind, 70, 306
limits.h, contents, 84
Lines
  reading, 202, 270
  writing, 390
LINTARGS, 4, 16, 79
Local time corrections, 32, 33, 308, 510
localtime, 72, 308
locking
  DOS considerations, 23
  XENIX version, differences from, 556, 45, 311
locking.h. See sys\locking.h
log function
  floating-point support, 25, 58, 316
log10 function
  floating-point support, 25, 58, 316
Logarithmic functions
  log, 316
  log10, 316
Long integers, conversion to strings, 323
Long pointers, 221
longjmp, 73, 318
lsearch, 70, 306
lseek
  XENIX version, differences from, 556, 54, 320
ltoa, 44, 323

Macros
  advantages over functions, 12
  arguments with side effects, 13, 43
  notational conventions, 6
  restrictions on use, 12
malloc, 60, 326
malloc.h, 62
malloc.h, contents, 84

Manifest constants, notational conventions, 6
Mask. See Permission mask
Math errors, 540
matherr, 18, 58, 328
math.h, 44, 60
math.h, contents, 84
max macro, 92
  _memavl, 60, 331
memccpy, 41, 333
memchr, 41, 335
memcmp, 41, 337
memcpy, 41, 339
memcmp, 41, 341
  _memmax, 60, 343
memmove, 41, 344
Memory allocation
  _fmalloc, 60, 214
  _fmsize, 60, 366
  _nmalloc, 60, 364
  _nmsize, 60, 366
malloc, 60, 125
  _expand, 60, 179
  _ffree, 60, 196
  _fheapchk, 60
  _fheapset, 60
  _fheapwalk, 60
  _fmalloc, 60, 214
  _fmsize, 60, 217
free, 60, 233
  _freect, 60, 235
halloc, 60, 276
hfree, 60, 278
include files, 62
malloc, 60, 326
  _memavl, 60, 331
  _memmax, 60
  _msize, 356
  _nfree, 60, 359
  _nheapchk, 60
  _nheapset, 60
  _nheapwalk, 60
  _nmalloc, 60, 364
  _nmsize, 60, 366
realloc, 60, 400
sbrk, 60, 413
stackavail, 60, 456, 31
models, use of with huge arrays and huge pointers, 26
Memory allocation
  _fheapchk, 204
Reference Index

Memory allocation (continued)
- fheapset, 205
- fheapwalk, 206
- memmax, 343
- nheapchk, 361
- nheapset, 362
- nheapwalk, 363
memory.h, 42
memory.h, contents, 85
memset, 41, 345
min macro, 92
Miscellaneous routines
- div, 160
- ldiv, 304
mkdir, 45, 347
mktemp, 45, 349
mktime, 72, 351
modf
- floating-point support, 25, 59, 352
Modification time, 522
movedata, 41, 354
- msize, 356

NDEBUG, 74, 80, 107
- NFILE constant, 88
- nfree, 60, 359
- nheapchk, 60, 361
- nheapset, 60, 362
- nheapwalk, 60, 363
- nmalloc, 60, 364
- nmsize, 60, 366
Nonlocal goto, 75, 318, 425
Notational conventions, 6
NULL
- constant, 88
- pointer, 50, 87
Null pointer, 88

O_BINARY, 22, 34
oflag. See Open flag
open
- argument-type-checking limitations, 17
- default translation mode, changing, 22
- default translation mode, overriding, 23
- XENIX version, differences from, 556, 54, 367, 428
Open flag, 367
Optional arguments, notational conventions, 7
- osmajor variable, 24, 34
- osminor variable, 24, 34
O_TEXT, 23
outp, 57, 371
Output. See I/O
OVERFLOW, 328, 540
Overlapping moves, 339
Overlay of parent process, 443

Parameters
- variable number, 73
- variable-length number, 524, 529
Parent process
- overlay, 443
- suspension, 443, 171, 443
Path names
- delimiters, 19, 20
- notational conventions, 6
Path-name conventions, 19
Permission mask, 514
Permission setting
- access, 97
- changing, 132
- umask, 514, 149, 367
perror, 18, 73, 372
PLOSS, 328, 540
Port I/O. See Console and port I/O
Portability, 19
Portability. See also Compatibility
Positioning file pointer, 200, 243, 246, 251, 320, 406, 499
pow
- floating-point support, 25, 59, 374
Predefined
- handles, 55
- stream pointers, 50
printf
- argument-type-checking limitations, 17, 48, 376
printf family
- floating-point support, 25
Printing. See Write operations
Process
- defined, 67
Process control
- abort, 66, 94
- atexit, 66, 111


Process control (continued)
  exec family, 66
  execl, 66, 171
  execlp, 66, 171
  execle, 66, 171
  execvp, 66, 171
  _exit, 66
  _exit, 176
  exit, 66
  exit, 176
  getpid, 66, 269
  include files, 68
  raise, 66, 396
  signal, 66, 432
  spawn family, 66
  spawnl, 66, 443
  spawnle, 66, 443
  spawnlp, 66, 443
  spawnlpe, 66, 443
  spawnv, 66, 443
  spawnve, 66, 443
  spawnvp, 66, 443
  spawnvpe, 66, 443
  system, 66, 494
Process ID, 269
  process.h, 68
  process.h, contents, 85
Processor calls
  FP_OFF, 66
  FP_SEG, 66
  include files, 66
  segread, 66
Program segment prefix, 35
Programming examples, notational conventions, 7
Pseudorandom integers, 397, 452
PSP. See Program segment prefix
  _psp variable, 35
ptrdiff_t, 87
putc, 48, 384
putch, 57, 386
putchar, 48, 384
putenv, 73, 387
puts, 48
putw, 48, 391
qsort, 70, 394
Quick sort, 394
raise, 66, 396
rand, 73, 397
Random Access, 200
Random access, 243
Random Access, 246
Random access, 251, 320, 406, 499
Random-number generator, 397, 452
read
  end-of-file condition, 19
  XENIX version, differences from, 557, 54, 398
Read access. See Permission setting
Read operations
  binary int value from stream, 271
  character from stdin, 198, 260
  character from stream, 198, 260
  characters from file, 398
  data items from stream, 231
  formatted, 152, 241, 415, 454
  from console
    checking for keystroke, 301
    with echo, 264, 128, 152, 262
  from port, 282
  line from stdin, 270
  line from stream, 202
realloc, 60, 400
Reallocation
  _expand, 179
  realloc, 400
Redirection, 51, 56, 237
Register, segment. See Segment registers
REGS type, 36, 82
Remainder function, 216
remove, 45, 402
rename, 45, 404
Return value on error. See Error returns
Reversing strings, 478
rewind, 48, 406
rmdir, 45, 408
rmtmp, 48, 410
Routines
  argument lists, variable-length
    va_arg, 73, 524
    va_end, 73, 524
    va_start, 73, 524
Reference Index

argument lists, variable-length (continued)
  vfprintf, 529
  vprintf, 529
  vsprintf, 529

math
  acos, 58, 99
  asin, 58, 105
  atan, 58, 109
  atan2, 58, 109
  bessel, 58, 118
  cabs, 58
  ceil, 58, 127
  _clear87, 58
  _control87, 58
  cos, 58, 144
  cosh, 58, 144
  dieetomsbin, 58
  dmsbintoeiee, 58
  errno, use of, 18
  error handling, 3, 18, 60
  exp, 58, 178
  fabs, 58, 182
  fieetomsbin, 58
  floor, 58, 211
  fmod, 58, 216
  fmsbintoeiee, 58
  _fpreset, 58
  frexp, 58, 240
  hypot, 58, 280
  include files, 60
  ldexp, 58, 303
  log, 58, 316
  log10, 58, 316
  matherr, 58, 328
  modf, 59, 352
  pow, 59, 374
  sin, 59, 436
  sinh, 59, 436
  sqrt, 59, 451
  _status87, 59
  tan, 59, 497
  tanh, 59, 497

miscellaneous
  abs, 73
  assert, 73, 107
  div, 73
  getenv, 73, 267
  include files, 74
  labs, 73
  ldiv, 73

miscellaneous (continued)
  longjmp, 73, 318
  perror, 73, 372
  putenv, 73, 387
  rand, 73, 397
  setjmp, 73, 425
  srand, 73, 452
  strerror, 467
  swab, 73, 492

Routines by category, 41

sbrk, 60, 413

scanf
  argument-type-checking limitations,
  17, 48, 415

scanf family
  floating-point support, 25
  scanf. See also scanf family

Scanning. See also Read operations

search.h, 70

search.h, contents, 86

Searching and sorting
  bsearch, 70, 120
  include files, 70
  lfind, 70, 306
  lsearch, 70, 306
  lsort, 70, 394

Seed, 452

Segment registers, obtaining values,
  421

segread, 66, 421

setbuf, 48, 52, 423

setjmp, 73, 425

setjmp.h, 74, 86

setmode, 23, 45, 428

setvbuf, 48, 52

share.h, 86

shflag. See Flag, sharing

Side effects in macro arguments, 13, 43

SIGFPE, 83, 432

SIGINT, 432

Signal
  raise, 396
  settings, child process, 174, 443

signal
  XENIX version, differences from,
  557, 66, 432

signal.h, 68, 87

sin
  floating-point support, 25, 59, 436
Sine, 436
SING, 328, 540
sinh
  floating-point support, 25, 59, 436
size_t, 87
Small capitals, use of, 8
sopen
  argument-type-checking limitations, 17
  DOS considerations, 23, 54
Sorting. See Searching and sorting
spawn family
  DOS considerations, 24
  path-name delimiters, 21
  spawn routines, differences between, 68, 66, 443
spawnl
  argument-type-checking limitations, 17, 66, 443
spawnl. See also spawn family
spawnle
  argument-type-checking limitations, 17, 66, 443
spawnle. See also spawn family
spawnlp
  argument-type-checking limitations, 17, 66, 443
spawnlp. See also spawn family
spawnlpe
  argument-type-checking limitations, 17, 66, 443
spawnlpe. See also spawn family
spawnv, 66, 443
spawnv. See also spawn family
spawnve, 66, 443
spawnve. See also spawn family
spawnvpe, 66, 443
sprintf
  argument-type-checking limitations, 17, 48, 449
sprintf. See also printf family
sqrt
  floating-point support, 25, 59, 451
Square-root function, 451
srand, 73, 452
SREGS type, 36, 82
sscanf
  argument-type-checking limitations, 17, 454
sscanf. See also scanf family
Stack checking, 15
Stack environment
  restoring, 318
  saving, 425
stackavail, 60, 456
Standard auxiliary. See stdaux
Standard error. See stderr
Standard input. See stdin
Standard output. See stdout
Standard print. See stdprn
Standard types
  complex, 36, 85
  div_t, 36
  DOSERROR, 36, 82, 162
  exception, 36, 85, 328
  FILE, 36, 89
  jmp_buf, 36
  ldiv_t, 36
  listed, 36
  REGS, 36, 82
  SREGS, 36, 82
  stat, 37, 90, 248, 458
  timeb, 37, 253
  time_t, 158
  tm, 37, 91, 273
  utimbuf, 37, 91, 522
stat
  XENIX version, differences from, 557, 45, 458
stat type, 37, 90, 248, 458
stat.h. See sys\stat.h
  _stat87, 59, 461
stdarg.h, contents, 87
stdaux
  buffering, 52
  default translation mode, overriding, 23
  file handle, 55
  translation mode, changing, 428, 50
stddef.h, contents, 87
stderr
  buffering, 52
  default translation mode, overriding, 23
  file handle, 55
  translation mode, changing, 428, 50
stdin
  buffering, 52
  default translation mode, overriding, 23
  file handle, 55
**Reference Index**

stdin (*continued*)
- translation mode, changing, 428, 50
- stdio.h, 50
- stdio.h, contents, 88
- stdlib.h, 43, 44, 74
- stdlib.h, contents, 89

stdout
- buffering, 52
- default translation mode, overriding, 23
- file handle, 55
- translation mode, changing, 428, 50

stderr
- buffering, 52
- default translation mode, overriding, 23
- file handle, 55
- translation mode, changing, 428, 50

`strcat`, 70, 463
`strchr`, 70, 463
`strcmp`, 70, 463
`strcpy`, 70, 463
`strcspn`, 70, 463
`strdup`, 70, 463
`Stream I/O`
- buffering, 52
- clearerr, 48, 138
- error handling, 19, 53
- fclose, 48, 183
- fcloseall, 48, 183
- fdopen, 48, 187
- feof, 48, 190
- ferror, 48, 192
- fflush, 48, 194
- fgetc, 48, 198
- fgetchar, 198
- fgetchar 48
- fgetpos, 48, 200
- fgets, 48, 202
- fileno, 48, 210
- flushall, 48, 212
- fopen, 48, 218
- fprintf, 48, 226
- fputc, 48, 228
- fputchar, 48, 228
- fputs, 48, 230, 390
- fread, 48, 231
- freopen, 48, 237
- fscanf, 48, 241
- fseek, 48, 243

Stream I/O (*continued*)
- fsetpos, 48, 246
- ftell, 48, 251
- fwrite, 48, 255
- getc, 48, 260
- getchar, 48, 260
- gets, 48, 270
- getw, 48, 271
- include files, 50
- printf, 48, 376
- putc, 48, 384
- putchar, 48, 384
- puts, 48
- putw, 48, 391
- rewind, 48, 406
- rmtmp, 48
- scanf, 48, 415
- setbuf, 48, 423
- setvbuf, 48
- sprintf, 48, 449
- sscanf, 48, 454
- tempnam, 48
- tmpfile, 48
- tmpnam, 48
- ungetc, 48, 516
- vfprintf, 48, 529
- vprintf, 48, 529
- vsprintf, 48, 529, 47, 48

Stream I/O. See also Console and port I/O

Stream pointer, 47

Streams
- appending, 188, 218, 237
- buffering, 423
- clearing errors, 138
- closing, 52, 183
- file handles for, 210
- file pointer positioning, 200, 243, 246, 251, 406
- formatted I/O, 226, 241, 376, 415, 449, 454, 529
- opening, 50, 187, 218
- reading
  - binary int value, 271
  - characters, 198, 260
  - data items, 231
  - lines, 202, 270
- reopening, 237
- rewinding, 406
- stdaux, 50
- stderr, 50

574
Streams (continued)
stdin, 50
stdout, 50
stderr, 50
translation mode, 188, 219, 237, 238
ungetting characters, 516
writing
  binary int value, 391
  characters, 228, 384
  data items, 255
  lines, 390
  strings, 230
strerror, 18, 70, 467
stricmp, 70, 463
String manipulation
  include files, 71
  strcat, 70, 463
  strchr, 70, 463
  strcmp, 70, 463
  strncmp, 70, 463
  strncpy, 70, 463
  strpbrk, 70, 463
  strstr, 70, 463
  strtok, 70, 463
  strncat, 70, 463
  strncmp, 70, 463
  strncpy, 70, 463
  strnicmp, 70, 463
  strnset, 70, 463
  strstr, 70, 463
  strncat, 70, 463
  strlen, 70, 469
  strlwr, 70, 470
  strncat, 70, 471
  strncmp, 70, 471
  strncpy, 70, 471
  strnicmp, 70, 471
  strlen, 70, 469
  strlwr, 70, 470
Subdirectory conventions, 19, 20
swab, 73, 492
Syntax conventions. See Notational
  conventions
  sys subdirectory, 20
  sys\locking.h, contents, 90
  sys\errlist, 33
  sys\errlist variable, 82, 372, 467
  sys\stat.h, contents, 90
  sys\timeb.h, 73
  sys\timeb.h, contents, 91
  sys\types.h, 73
  sys\types.h, contents, 91
  sys\utime.h, 73
  sys\utime.h, contents, 91
  sys\nerr, 33
  sys\nerr variable, 372, 467
  sysat.h, 46
  system
    path-name delimiters, 21
    XENIX version, differences from, 558, 66, 494
System calls. See DOS system calls
System time. See Time

- tan
  - floating-point support, 25, 59, 497
- Tangent, 497
- tanh
  - floating-point support, 25, 59, 497
  - tell, 54, 499
- tempnam, 48, 501
- Terminal capabilities, 294
- Text mode, 22, 34, 188, 219, 237, 238, 367, 428
- Time
  - computing time differences, 158
  - conversion from
    - long integer to string, 154
    - long integer to structure, 308
    - structure to string, 103
  - correcting for local time, 308
  - global variables, setting, 510
  - obtaining, 253, 504
- routines
  - asctime, 72, 103
  - clock, 72, 139
  - ctime, 72, 154
  - difftime, 72, 158
  - ftime, 72, 253
  - gmtime, 72, 273
  - include files, 72, 73
  - localtime, 72, 308
  - mktime, 72, 351
  - time, 72, 504
  - tzset, 72, 510
  - utime, 72, 522
- time, 72, 504
- timeb type, 37, 253
- timeb.h. See sys\timeb.h
- time.h, 72, 73
- time.h, contents, 91
- time_t type, 158
- timezone variable, 32, 510
- TLOSS, 328, 540
- tm type, 37, 91, 273
- tmpfile, 48
- tmpnam, 48, 501
- toascii, 42, 507
- Tokens, finding in strings, 489
- tolower
  - function version, use of, 43
  - side effects, 43
- toupper
  - function version, use of, 43
  - side effects, 43
  - toupper, 42
  - toupper, 42
  - toupper, 507
  - toupper, 507
- Translation mode. See Binary mode or Text mode
- Trigonometric functions
  - acos, 99
  - asin, 105
  - atan, 109
  - atan2, 109
  - cos, 144
  - cosh, 144
  - hypot, 280
  - sin, 436
  - sinh, 436
  - tan, 497
  - tanh, 497
- Type checking, arguments. See
  - Argument type checking
- Types
  - predefined. See Standard types
  - types.h. See sys\types.h
- TZ environment variable, 32
- TZ environment variable, default value, 33
- TZ variable, 308, 510
- tzname variable, 32, 510
- tzset, 72, 510
- ultoa, 44
- umask
  - XENIX version, differences from, 559, 45, 514
- UNDERFLOW, 328, 540
- ungetc, 48, 516
- ungetch, 57, 518
- UNIX operating system, 543
- unlink
  - XENIX version, differences from, 559, 45, 520
- Update, 188, 218, 237
- utimbuf type, 37, 91, 522
utime
  XENIX version, differences from, 559, 72, 522
utime.h. See sys\utime.h

v2tov3.h, contents, 92
va_arg, 73, 524
va_end, 73, 524
varargs.h, contents, 92

Variable
global
  _ ambksz, 31
  common library, use in, 547
daylight, 32, 510
  _ doserrno, 33
environ, 35, 267, 387
errno, 33, 82, 372, 467, 537
  _ fmode, 34
  _ osmajor, 34
  _ osminor, 34
  _ psp, 35
sys_errlist, 33, 82, 372, 467
sys_nerr, 33, 372, 467
timezone, 32, 510
tzname, 32, 510, 31

Variable, global. See Global variables
va_start, 73, 524

Version number (DOS), 34
vfprintf, 48, 529
vprintf, 48, 529
vsprintf, 48, 529

Write operations (continued)
to console, 146, 148, 386
to port, 371

XENIX operating system, 543

y0. See Bessel
y1. See Bessel
yn. See Bessel"

Word. See Binary int
WORDREGS type, 82
write
  XENIX version, differences from, 559, 54, 532

Write access. See Permission setting
Write operations
  binary int value to stream, 391
  character
to console, 518
to file, 532
to stdout, 228, 384
to stream, 228, 384, 516
data items from stream, 255
  formatted, 146, 226, 376, 449, 529
  line to stream, 390
  string to stream, 230