CodeWarrior®
MSL C Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Mail order	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
America Online	keyword: `Metrowerks`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction

This reference contains a description of the ANSI library and extended libraries bundled with Metrowerks C.

Organization of Files

The C headers files are organized alphabetically. Items within a header file are also listed in alphabetical order. Whenever possible, sample code has been included to demonstrate the use of each function.

ANSI C Standard

The ANSI C Standard Library included with Metrowerks CodeWarrior follows the specifications in the ANSI: Programming Language C / X3.159.1989 document. The functions, variables and macros available in this library can be used transparently by both C and C++ programs.

The ANSI C Library and Apple Macintosh

Some functions in the ANSI C Library are not fully operational on the Macintosh environment because they are meant to be used in a character-based user interface instead of the Macintosh computer's graphical user interface. While these functions are available, they may not work as you expect them to. Such inconsistencies between the ANSI C Standard and the Metrowerks implementation are noted in a function's description.

Except where noted, ANSI C Library functions use C character strings, not Pascal character strings.

Console I/O and the Macintosh

The ANSI Standard Library assumes interactive console I/O (the `stdin`, `stderr`, and `stdout` streams) is always open. Many of the functions in this library were originally designed to be used on a character-oriented user interface, not the graphical user interface of a Macintosh computer. These header files contain functions that help you run character-oriented programs on a Macintosh:

- `console.h` declares `ccommand()`, which displays a dialog that lets you enter command-line arguments
- `SIOUX.h` is part of the SIOUX package, which creates a window that's much like a dumb terminal or TTY. Your program uses that window whenever your program refers to `stdin`, `stdout`, `stderr`, `cin`, `cout`, or `cerr`.
- `unix.h unistd.h, stat.h fcntl.h and utsname.h` declare several functions common on UNIX systems that are not part of the ANSI standard.

2

alloca.h

This header defines one function `alloca()` which lets you allocate memory quickly on a Power Macintosh.

alloca

Description Allocates memory quickly onthe stack.

WARNING! This function is not available with the 68K Macintosh compiler.

Prototype

```
#include <alloca.h>
void *alloca(size_t nbytes);
```

Remarks This function returns a pointer to a block of memory that is `nbytes` long. The block is on the function's stack. This function works quickly since it decrements the current stack pointer. When your function exits, it automatically releases the storage.

If you use `alloca()` to allocate a lot of storage, be sure to increase the Stack Size for your project in the Project preferences panel.

Return If it is successful, `alloca()` returns a pointer to a block of memory. If it encounters an error, `alloca()` returns `NULL`.

See Also `stdlib.h: calloc(), free(), malloc(), realloc()`

3

assert.h

The `assert.h` header file provides a debugging macro that outputs a diagnostic message and stops the program if a test fails.

Diagnostics

The `assert.h` header file provides a debugging macro that outputs a diagnostic message and stops the program if a test fails.

assert

Description Abort a program if a test is false.

Prototype

```
#include <assert.h>
void assert(int expression);
```

Remarks If `expression` is false the `assert()` macro outputs a diagnostic message to `stderr` and calls `abort()`. The diagnostic message has the form

```
file: line test -- assertion failed

abort -- terminating
```

where

- `file` is the source file,
- `line` is the line number, and
- `test` is the failed expression.

To turn off the assert() macros, place a `#define NDEBUG` (no debugging) directive before the `#include <assert.h>` directive.

See Also **stdlib.h**: abort(), **stdio.h**

Listing 3.1 Example of assert() usage.

```
#undef NDEBUG
    /* Make sure that assert() is enabled */
#include <assert.h>
#include <stdio.h>
void main(void)
{
  int x = 100, y = 5;
  printf("assert test.\n");

/*This assert will output a message and abort the program */
  assert(x > 1000);
  printf("This will not execute if NDEBUG is undefined\n");
}
```

```
/* Output:
assert test.
foo.c:12 x > 1000 -- assertion failed
abort -- terminating
*/
```

4

console.h

This header file contains one function, `ccommand()`, which helps you port a program that relies on command-line arguments.

Using Command-line Arguments

This header file contains one function, `ccommand()`, which helps you port a program that relies on command-line arguments.

NOTE: If you're porting a UNIX or DOS program, you might also need the functions in `unix.h` and `SIOUX.h`.

ccommand

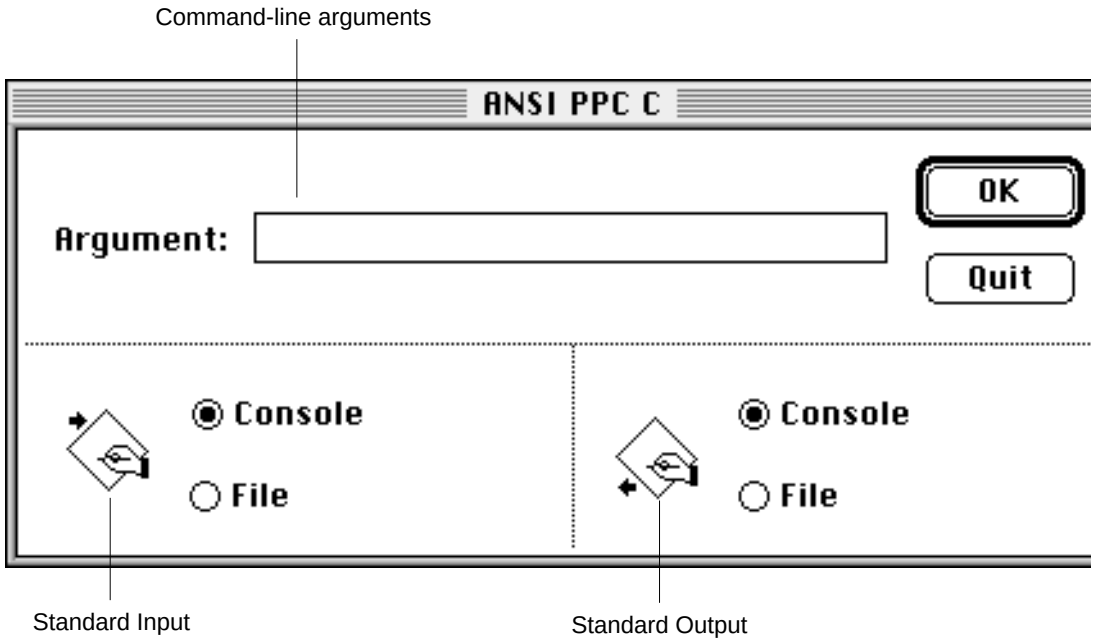
Description Lets you enter command-line arguments for a SIOUX program.

Prototype

```
#include <console.h>
int ccommand(char ***);
```

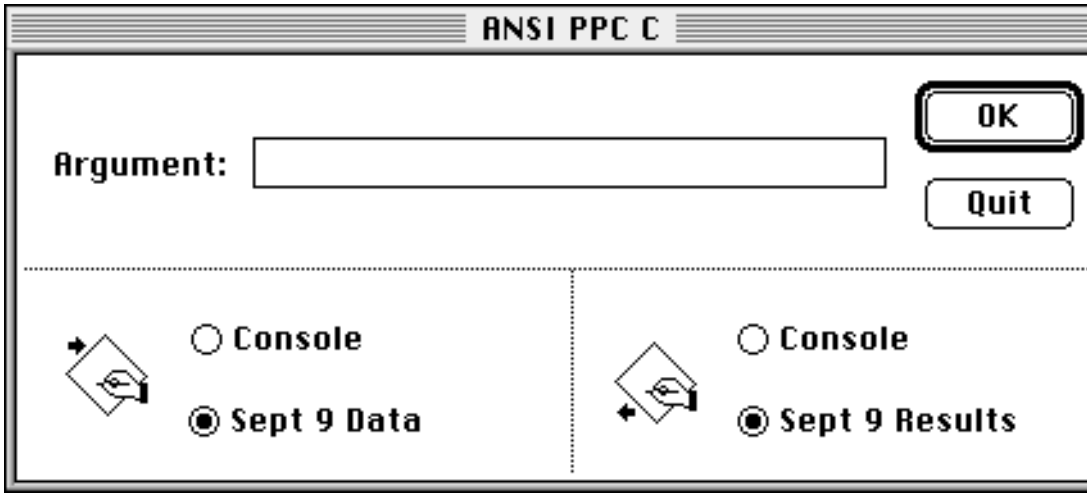
Remarks This function displays a dialog that lets you enter arguments and redirect standard input and output, as shown in "The ccommand dialog" on page 18.

Figure 4.1 The ccommand dialog



Enter the command-line arguments in the Argument field. Choose where your program directs standard input and output with the buttons below the field: the buttons on the left are for standard input and the buttons on the right are for standard output. If you choose Console, the program reads from or write to a SIOUX window. If you choose File, `ccommand()` displays a standard file dialog which lets you choose a file to read from or write to. After you choose a file, its name replaces the word *File*, as shown in "Redirecting input and output to files" on page 19.

Figure 4.2 Redirecting input and output to files



The function `ccommand()` returns an integer and takes one parameter which is a pointer to an array of strings. It fills the array with the arguments you entered in the dialog and returns the number of arguments you entered. As in UNIX or DOS, the first argument, the argument in element 0, is the name of the program. "Example of ccommand()" on page 19 has an example of command line usage

Return This function returns the number of arguments you entered.

See Also `SIOUX.h`

Listing 4.1 Example of ccommand()

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

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

  argc = ccommand(&argv);
```

```
  for (i = 0; i < argc; i++)
      printf("%d. %s\n", i, argv[i]);
  return 0;
}
```

5

ctype.h

The `ctype.h` header file supplies macros and functions for testing and manipulation of character type

Character testing and case conversion

The `ctype.h` header file supplies macros for testing character type and for converting alphabetic characters to uppercase or lowercase. The `ctype.h` macros support ASCII characters (`0x00` to `0x7F`), and the `EOF` value. These macros are not defined for the Apple Macintosh Extended character set (`0x80` to `0xFF`).

isalnum

Description Determine character type.

Prototype

```
#include <ctype.h>
int isalnum(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage see “Character testing functions example” on page 22

Return “Character testing functions” on page 22 describes what the character testing functions return.

Table 5.1 Character testing functions

This function	Returns true if c is
isalnum(c)	Alphanumeric: `[a-z]`, `[A-Z]`, `[0-9]`
`isalpha(c)`	Alphabetic: `[a-z]`, `[A-Z]`.
`iscntrl(c)`	The delete character (`0x7F`) or an ordinary control character from `0x00` to `0x1F`.
isdigit(c)	A numeric character: `[0-9]`.
isgraph(c)	A non-space printing character from the exclamation (`0x21`) to the tilde (`0x7E`).
islower(c)	A lowercase letter: `[a-z]`.
`isprint(c)`	A printable character from space (`0x20`) to tilde (`0x7E`).
ispunct(c)	A punctuation character. A punctuation character is neither a control nor an alphanumeric character.
isspace(c)	A space, tab, return, new line, vertical tab, or form feed.
isupper(c)	An uppercase letter: `[A-Z]`.
isxdigit(c)	A hexadecimal digit `[0-9]`, `[A-F]`, or `[a-f]`.

See Also `tolower(), toupper()`

Listing 5.1 Character testing functions example

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

void main(void)
{
  int a = 'F', b = '6', c = '#', d = 9;
```

```
  printf("isalnum for %c: %d\n", b, isalnum(b));
  printf("isalpha for %c: %d\n", a, isalpha(a));
  printf("iscntrl for %c: %d\n", d, iscntrl(d));
  printf("isdigit for %c: %d\n", d, isdigit(d));
  printf("isgraph for %c: %d\n", d, isgraph(d));
  printf("islower for %c: %d\n", a, islower(a));
  printf("isprint for %c: %d\n", d, isprint(d));
  printf("ispunct for %c: %d\n", c, ispunct(c));
  printf("isspace for %c: %d\n", d, isspace(d));
  printf("isupper for %c: %d\n", b, isupper(b));
  printf("isxdigit for %c: %d\n", a, isxdigit(a));
}
```

```
Output:
isalnum for 6: 32
isalpha for F: 2
iscntrl for : 64
isdigit for : 0
isgraph for : 0
islower for F: 0
isprint for : 0
ispunct for #: 8
isspace for : 64
isupper for 6: 0
isxdigit for F: 1
```

isalpha

Description Determine character type.

Prototype

```
#include <ctype.h>
int isalpha(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.2 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

iscntrl

Description Determine character type.

Prototype

```
#include <ctype.h>
int iscntrl(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.3 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

isdigit

Description Determine character type.

Prototype

```
#include <ctype.h>
int isdigit(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.4 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

isgraph

Description Determine character type.

Prototype

```
#include <ctype.h>
int isgraph(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.5 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

islower

Description Determine character type.

Prototype

```
#include <ctype.h>
int islower(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.6 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

isprint

Description Determine character type.

Prototype

```
#include <ctype.h>
int isprint(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.7 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

ispunct

Description Determine character type.

Prototype
```
#include <ctype.h>
int ispunct(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.8 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

isspace

Description Determine character type.

Prototype
```
#include <ctype.h>
int isspace(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.9 For example usage

```
For example usage see "Character testing functions example" on
```

page 22

isupper

Description Determine character type.

Prototype

```
#include <ctype.h>
int isupper(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`.

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.10 For example usage

```
For example usage see "Character testing functions example" on
page 22
```

isxdigit

Description Determine character type.

Prototype

```
#include <ctype.h>
int isxdigit(int c);
```

Remarks This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage see "Character testing functions example" on page 22

Return "Character testing functions" on page 22 describes what the character testing functions return.

Listing 5.11 For example usage

```
For example usage see “Character testing functions example” on
page 22
```

tolower

Description Character conversion macro.

Prototype

```
#include <ctype.h>
int tolower(int c);
```

Remarks The `tolower()` macro converts an uppercase letter to its lowercase equivalent. Non-uppercase characters are returned unchanged. For example usage see “Example of tolower(), toupper() usage.” on page 29

Return `tolower()` returns the lowercase equivalent of uppercase letters and returns all other characters unchanged.

See Also `isalpha(), toupper()`

Listing 5.12 Example of tolower(), toupper() usage.

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

void main(void)
{
  static char s[] =
        "** DELICIOUS! lovely? delightful **";
  int i;

  for (i = 0; s[i]; i++)
    putchar(tolower(s[i]));
  putchar('\n');
```

```
  for (i = 0; s[i]; i++)
    putchar(toupper(s[i]));
  putchar('\n');
}
```

```
Output:
** delicious! lovely? delightful **
** DELICIOUS! LOVELY? DELIGHTFUL **
```

toupper

Description Character conversion macro.

Prototype

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

Remarks The `toupper()` macro converts a lowercase letter to its uppercase equivalent and returns all other characters unchanged.

Return `toupper()` returns the uppercase equivalent of a lowercase letter and returns all other characters unchanged.

Listing 5.13 For example usage

see “Example of tolower(), toupper() usage.” on page 29

6

errno.h

The `errno.h` header file provides the global error code variable `errno`.

Runtime error codes

Description

The `errno.h` header file provides the global error code variable `errno`.

Prototype

```
#include <errno.h>
extern int errno;
```

Most functions in the standard library return a special value when an error occurs. Often the programmer needs to know about the nature of the error. Some functions provide detailed error information by assigning a value to the global variable `errno`. The `errno` variable is declared in the `errno.h` header file. See “Error number defintiions.” on page 32

For example, the `double pow(double x, double y)` function in `math.h` computes the floating point value of x^y. The expression 0^n cannot be expressed using the `double` type if n equal to or less than zero. If the `x` argument is 0.0 and the `y` argument is equal or less than 0.0, the `pow()` function assigns the `EDOM` (Domain error) value to `errno` and returns 0.0.

The `errno` variable is not cleared when a function call is successful; its value is changed only when a function that uses `errno` returns its own error value. It is the programmer's responsibility to assign 0 to `errno` before calling a function that uses it. For example usage see “errno example” on page 32

Table 6.1 Error number defintiions.

errno value	Description
EDOM	Domain error. The arguments passed to the function are not within a legal domain. This macro is defined as a nonzero value in errno.h.
ERANGE	Range error. The function cannot return a value represented by its type. This macro is defined as a nonzero value in errno.h.
nonzero value	Used by some stdlib.h and stdio.h functions.

Figure 6.1 errno example

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

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

  printf("Enter two floating point values.\n");
  scanf("%lf %lf", &x, &y);
  errno = 0;// reset errno before doing operation
  result = pow(x, y);
  if (errno == EDOM)
    printf("Domain error!\n");
  else
    printf("%f to the power of %f is %f.\n",
          x, y, result);
}
```

```
/* Output:
Enter two floating point values.
1.2
```

```
3.4
1.200000 to the power of 3.400000 is 1.858730.
*/
```

fcntl.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

creat

Description Create a new file or overwrite an existing file.

Prototype

```
#include <fcntl.h>
int creat(const char *filename, int mode);
```

Remarks This function creates a file named `filename` you can write to. If the file does not exist, `creat()` creates it. If the file already exists, `creat()` overwrites it. The function ignores the argument mode.

This function call:

```
creat(path, mode);
```

is equivalent to this function call:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

Return If it's successful, `creat()` returns the file description for the created file. If it encounters an error, it returns `-1`.

See Also `stdio.h: fopen(); unix.h: fdopen(), open(), close()`

Listing 7.1 Example of creat() usage.

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

void main(void)
{
  int fd;

  fd = creat("Jeff:Documents:mytest", 0);
    /* Creates a new file named mytest in the folder
       Documents on the volume Akbar. */
  write(fd, "Hello world!\n", 13);
  close(fd);
}
```

fcntl

Description Manipulates a file descriptor.

Prototype

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
```

Remarks This function performs the command specified in `cmd` on the file descriptor `fildes`.

In the Metrowerks ANSI library, `fcntl()` can perform only one command, `F_DUPFD`. This command returns a duplicate file descriptor for the file that `fildes` refers to. You must include a third argument in the function call. The new file descriptor is the lowest

available file descriptor that is greater than or equal to the third argument.

Return If it is successful, `fcntl()` returns a file descriptor. If it encounters an error, `fcntl()` returns `-1`.

See Also `unix.h: fileno(), open(), fdopen()`

Listing 7.2 Example of fcntl() usage.

```
#include <unix.h>

void main(void)
{
  int fd1, fd2;

  fd1 = open("mytest", O_WRONLY | O_CREAT);

  write(fd1, "Hello world!\n", 13);
    /* Write to the original file descriptor.          */

  fd2 = fcntl(fd1, F_DUPFD, 0);
    /* Create a duplicate file descriptor.             */

  write(fd2, "How are you doing?\n", 19);
    /* Write to the duplicate file descriptor.         */

  close(fd2);
}
```

```
/*ReslutAfter you run this program,
the file mytest contains the following:
Hello world!
How are you doing?
*/
```

open

Description `Opens a file and returns it's id.`

Prototype

```
int open(const char *path, int oflag);
```

Remarks The function `open()` opens a file for system level input and output. and is used with the UNIX style functions `read`() and `write`().

Table 7.1 Legal file opening modes with open()

Mode	Description
O_RDWR	Open the file for both read and write
O_RDONLY	Open the file for read only
O_WRONLY	Open the file for write only
O_APPEND	Open the file at the end of file for appending
O_CREAT	Create the file if it doesn't exist
O_EXCL	Do not create the file if the file already exists.
O_TRUNC	Truncate the file after opening it.
O_NRESOLVE	Don't resolve any aliases.
O_ALIAS	`Open alias file (if the file is an alias`).
O_RSRC	Open the resource fork
O_BINARY	Open the file in binary mode (default is text mode).
F_DUPFD	Return a duplicate file descriptor.

Return open() returns the file id as an integer value.

See Also `unistd.h: close(), lseek(), read() and write()`

Listing 7.3 Example of open() usage:

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

#define SIZE FILENAME_MAX
#define MAX 1024

char fname[SIZE] = "DonQ.txt";

void main(void)
{
  int fdes;
  char temp[MAX];
  char *Don =
"In a certain corner of la Mancha, the name of\n
which I do not choose to remember, there lived\n\
one of those country gentlemen, who adorn their\n\
halls with rusty lance and worm-eaten targets.";

  /* NULL terminate temp array for printf */
  memset(temp, '\0', MAX);

  /* open a file */
  if((fdes = open(fname,O_RDWR|O_TRUNC|O_BINARY ))==1)
  {
    printf("Can not open %s", fname);
    exit( EXIT_FAILURE);
  }

  /* write to a file  */
  if( write(fdes, Don, strlen(Don)) == -1)
```

```
  {
    printf("Write Error");
    exit( EXIT_FAILURE );
  }

  /* move to beginning of file for read */
  if( lseek( fdes, 0L, SEEK_SET ) == -1L)
  {
    printf("Seek Error");
    exit( EXIT_FAILURE );
  }

  /* read the file  */
  if( read( fdes, temp, MAX ) == 0)
  {
    printf("Read Error");
    exit( EXIT_FAILURE);
  }

  /* close the file */
  if(close(fdes))
  {
    printf("File Closing Error");
    exit( EXIT_FAILURE );
  }

   puts(temp);

}
```

```
In a certain corner of la Mancha, the name of
which I do not choose to remember,there lived
one of those country gentlemen, who adorn their
halls with rusty lance  and worm-eaten targets.
```

8

float.h

The `float.h` header file macros specify the characteristics of floating point number representation for `float`, `double` and `long double` types.

Floating point number characteristics

The `float.h` header file macros specify the characteristics of floating point number representation for `float`, `double` and `long double` types.

"Floating point characteristics" on page 41 lists the macros defined in `float.h`. Macros beginning with `FLT` apply to the `float` type; `DBL`, the `double` type; and `LDBL`, the `long double` type.

The `FLT_RADIX` macro specifies the radix of exponent representation.

The `FLT_ROUNDS` specifies the rounding mode. Metrowerks C rounds towards positive infinity.

Table 8.1 Floating point characteristics

Macro	Description
FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG	The number of base `FLT_RADIX` digits in the significand.
FLT_DIG, DBL_DIG, LDBL_DIG	The decimal digit precision.

Table 8.1 Floating point characteristics

FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP	The smallest negative integer exponent that `FLT_RADIX` can be raised to and still be expressible.
FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP	The smallest negative integer exponent that 10 can be raised to and still be expressible.
FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP	The largest positive integer exponent that `FLT_RADIX` can be raised to and still be expressible.
FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP	The largest positive integer exponent that 10 can be raised to and still be expressible.
FLT_MIN, DBL_MIN, LDBL_MIN	The smallest positive floating point value.
FLT_MAX, DBL_MAX, LDBL_MAX	The largest floating point value.
FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON	The smallest fraction expressible.

9

limits.h

The `limits.h` header file macros describe the maximum and minimum values of integral types.

Integral type limits

The `limits.h` header file macros describe the maximum and minimum values of integral types. “Integral limits” on page 43 describes the macros.

Table 9.1 Integral limits

Macro	Description
CHAR_BIT	Number of bits of smallest object that is not a bit field.
CHAR_MAX	Maximum value for an object of type `char`.
CHAR_MIN	Minimum value for an object of type `char`.
SCHAR_MAX	Maximum value for an object of type `signed char`.
SCHAR_MIN	Minimum value for an object of type `signed char`.
UCHAR_MAX	Maximum value for an object of type `unsigned char`.
SHRT_MAX	Maximum value for an object of type `short int`.
SHRT_MIN	Minimum value for an object of type `short int`.

USHRT_MAX	Maximum value for an object of type `unsigned short int`.
INT_MAX	Maximum value for an object of type `int`.
INT_MIN	Minimum value for an object of type `int`.
LONG_MAX	Maximum value for an object of type `long int`.
LONG_MIN	Minimum value for an object of type `long int`.
ULONG_MAX	Maximum value for an object of type `unsigned long int`.

10

locale.h

The `locale.h` header file provides facilities for handling different character sets and numeric and monetary formats.

Locale specification

The ANSI C Standard specifies that certain aspects of the C compiler are adaptable to different geographic locales. The `locale.h` header file provides facilities for handling different character sets and numeric and monetary formats. Metrowerks C supports only the `"C"` locale.

The `lconv` structure, defined in `locale.h`, specifies numeric and monetary formatting characteristics for converting numeric values to character strings. A call to `localeconv()` will return a pointer to an `lconv` structure containing the settings for the `"C"` locale ("lconv structure and contents returned by localeconv()" on page 45). An `lconv` member is assigned the `CHAR_MAX` value (defined in `limits.h`) if it is not applicable to the current locale.

Listing 10.1 lconv structure and contents returned by localeconv()

```
struct lconv {
  char *currency_symbol;
  char *int_curr_symbol;
  char *mon_decimal_point;
  char *mon_grouping;
  char *mon_thousands_sep;
  char *negative_sign;
  char *positive_sign;
  char frac_digits;
  char int_frac_digits;
  char n_cs_precedes;
```

```
char n_sep_by_space;
char n_sign_posn;
char p_cs_precedes;
char p_sep_by_space;
char p_sign_posn;
char *decimal_point;
char *grouping;
char *thousands_sep;
};
```

localeconv

Description Return the `lconv` settings for the current locale.

Prototype

```
#include <locale.h>
struct lconv *localeconv(void);
```

Return `localeconv()` returns a pointer to an `lconv` structure for the `"C"` locale. Refer to Figure 1.

setlocale

Description Query or set locale information for the C compiler.

Prototype

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Remarks The `category` argument specifies the part of the C compiler to query or set.

The argument can have one of six values defined as macros in `locale.h`: `LC_ALL` for all aspects, `LC_COLLATE` for the collating function `strcoll()`, `LC_CTYPE` for `ctype.h` functions and the multibyte conversion functions in `stdlib.h`, `LC_MONETARY` for monetary formatting, `LC_NUMERIC` for numeric formatting, and `LC_TIME` for time and date formatting.

If the `locale` argument is a null pointer or an empty string, a query is made. The `setlocale()` function returns a pointer to a character string indicating which locale the specified compiler part is set to. The Metrowerks C compiler supports the `"C"` locale.

Attempting to set a part of the Metrowerks C compiler's locale will have no effect.

See Also `type.h, stdlib.h, string.h: strcoll()`

11

math.h

The `math.h` header file provides floating point mathematical and conversion functions.

Floating point mathematics

The `HUGE_VAL` macro, defined in `math.h`, is returned as an error value by the `strtod()` function. Refer to the `stdlib.h` header for information on `strtod()`.

Some `math.h` functions use the `errno` global variable to indicate an error condition. In particular, many functions set `errno` to `EDOM` when an argument is beyond a legal domain. Refer to the `errno.h` header for information on `errno`.

acos

Description Arccosine function.

Prototype

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

Remarks This function computes the arc values of cosine, sine, and tangent.

The function `acos()` sets `errno` to `EDOM` if the argument is not in the range of -1 to +1. See “Example of acos(), asin(), atan(), atan2() usage.” on page 51 for example usage.

Return `acos()` returns the arccosine of the argument `x` in radians. If the argument to `acos()` is not in the range of -1 to +1, the global variable `errno` is set to `EDOM` and returns 0.

See Also `cos()`, **errno.h**

asin

Description Arcsine function.

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

Remarks This function computes the arc values of sine.

The function `asin()` sets `errno` to `EDOM` if the argument is not in the range of -1 to +1. See "Example of acos(), asin(), atan(), atan2() usage." on page 51 for example usage.

Return The function `asin()` returns the arcsine of `x` in radians. If the argument to `asin()` is not in the range of -1 to +1, the global variable `errno` is set to `EDOM` and returns 0.

See Also `sin()`, **errno.h**

atan

Description Arctangent function.

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

Remarks This function computes the value of the arc tangent of the argment. See "Example of acos(), asin(), atan(), atan2() usage." on page 51 for example usage.

Return The function `atan()` returns the arc tangent of the argument `x` in the range

[-π/2, +π/2] radians.

See Also `tan()`, **`errno.h`**

atan2

Description Arctangent function.

Prototype

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

Remarks This function computes the value of the tangent of `x/y` using the signs of both arguments. See "Example of acos(), asin(), atan(), atan2() usage." on page 51 for example usage.

Return The function `atan2()` returns the arc tangent of `y/x` in the range [-π, +π] radians.

See Also **`math.h`**: `tan()`, **`errno.h`**

Listing 11.1 Example of acos(), asin(), atan(), atan2() usage.

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

void main(void)
{
  double x = 0.5, y = -1.0;

  printf("arccos (%f) = %f\n", x, acos(x));
  printf("arcsin (%f) = %f\n", x, asin(x));
  printf("arctan (%f) = %f\n", x, atan(x));
  printf("arctan (%f / %f) = %f\n", y, x, atan2(y, x));
}
```

```
Output:
arccos (0.500000) = 1.047198
arcsin (0.500000) = 0.523599
```

```
arctan (0.500000) = 0.463648
arctan (-1.000000 / 0.500000) = -1.107149
```

ceil

Description Compute the smallest integer not less than *x*.

Prototype

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

Return `ceil()` returns the smallest integer not less than `x`.

See Also **`math.h`**: `floor(), fmod(), fabs()`

Listing 11.2 Example of ceil() usage.

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

void main(void)
{
  double x = 100.001, y = 9.99;

  printf("The ceiling of %f is %f.\n", x, ceil(x));
  printf("The ceiling of %f is %f.\n", y, ceil(y));

}
```

```
Output:
The ceiling of 100.001000 is 101.000000.
The ceiling of 9.990000 is 10.000000.
```

cos

Description Compute cosine.

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

Return `cos()` returns the cosine of `x`. `x` is measured in radians.

See Also **`math.h`**: `sin()`, `tan()`

Listing 11.3 Example of cos() usage

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

void main(void)
{
  double x = 0.0;
  printf("The cosine of %f is %f.\n", x, cos(x));

}
```

```
Output:
The cosine of 0.000000 is 1.000000.
```

cosh

Description Compute the hyperbolic cosine.

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

Return `cosh()` returns the hyperbolic cosine of `x`.

See Also **math.h**: sinh(), tanh()

Listing 11.4 cosh() example

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

void main(void)
{
  double x = 0.0;

  printf("Hyperbolic cosine of %f is %f.\n",x,cosh(x));
}
```

```
Output:
Hyperbolic cosine of 0.000000 is 1.000000.
```

exp

Description Compute e^x.

Prototype

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

Return exp() returns *e*x, where *e* is the natural logarithm base value.

See Also **math.h**: log(), log10()

Listing 11.5 exp() example

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

void main(void)
{
```

```
  double x = 4.0;
  printf("The natural logarithm base e raised to the\n");
  printf("power of %f is %f.\n", x, exp(x));
}
```

```
Output:
The natural logarithm base e raised to the
power of 4.000000 is 54.598150.
```

fabs

Description Compute the floating point absolute value.

Prototype

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

Return `fabs()` returns the absolute value of `x`.

See Also **`math.h`**: `floor()`, `ceil()`, `fmod()`

Listing 11.6 fabs() example

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

void main(void)
{
  double s = -5.0, t = 5.0;
  printf("Absolute value of %f is %f.\n", s, fabs(s));
  printf("Absolute value of %f is %f.\n", t, fabs(t));
}
```

```
Output:
Absolute value of -5.000000 is 5.000000.
Absolute value of 5.000000 is 5.000000.
```

floor

Description Compute the largest integer not greater than *x*.

Prototype

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

Return `floor()` returns the largest integer not greater than `x`.

See Also `ceil()`, `fmod()`, `fabs()`

Listing 11.7 floor() example

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

void main(void)
{
  double x = 12.03, y = 10.999;

  printf("Floor value of %f is %f.\n", x, floor(x));
  printf("Floor value of %f is %f.\n", y, floor(y));

}
```

```
Output:
Floor value of 12.030000 is 12.000000.
Floor value of 10.999000 is 10.000000.
```

fmod

Description Return the floating point remainder of *x* / *y*.

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

Return `fmod()` returns, when possible, the value *f* such that `x` = *i* `y` + *f* for some integer *i*, and |*f*| < |`y`|. The sign of *f* matches the sign of `x`.

See Also `floor(), ceil(), fmod(), fabs()`

Listing 11.8 Example of fmod() usage.

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

void main(void)
{
  double x = -54.4, y = 10.0;
  printf("Remainder of %f / %f = %f.\n",
         x, y, fmod(x, y));
}
```

```
Output:
Remainder of -54.400000 / 10.000000 = -4.400000.
```

frexp

Description Extract the mantissa and exponent.

Prototype
```
#include <math.h>
double frexp(double value, int *exp);
```

Remarks The `frexp()` function extracts the mantissa and exponent of `value` based on the formula $x*2^n$, where the mantissa is $0.5 \leq |x| < 1.0$ and n is an integer exponent.

Return `frexp()` returns the `double` mantissa of `value`. It stores the integer exponent value at the address referenced by `exp`.

See Also `ldexp(), modf()`

Listing 11.9 frexp() example

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

void main(void)
{
  double m, value = 12.0;
  int e;

  m = frexp(value, &e);

  printf("%f = %f * 2 to the power of %d.\n",value, m, e);
}
```

```
Output:
12.000000 = 0.750000 * 2 to the power of 4.
```

ldexp

Description Compute a value from a mantissa and exponent.

Prototype

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

Remarks The `ldexp()` function computes `x` * 2`exp`. This function can be used to construct a `double` value from the values returned by the `frexp()` function.

Return `ldexp()` returns `x` * 2`exp`.

See Also `frexp(), modf()`

Listing 11.10 Example of ldexp() usage.

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

void main(void)
{
  double value, x = 0.75;
  int e = 4;

  value = ldexp(x, e);

  printf("%f * 2 to the power of %d is %f.\n",x, e, value);
}
```

```
Output:
0.750000 * 2 to the power of 4 is 12.000000.
```

log

Description Compute the natural and base 10 logarithms.

Prototype
```
#include <math.h>
double log(double x);
double log10(double x);
```

Return `log()` returns $\log_e x$. If `x` < 0 the `log()` assigns `EDOM` to `errno`.

See Also exp(), **errno.h**

Listing 11.11 log(), log10() example

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

void main(void)
{
  double x = 100.0;

  printf("The natural logarithm of %f is %f\n",x, log(x));
  printf("The base 10 logarithm of %f is %f\n",x, log10(x));
}
```

```
Output:
The natural logarithm of 100.000000 is 4.605170
The base 10 logarithm of 100.000000 is 2.000000
```

log10

Description Compute the base 10 logarithms.

Prototype

```
#include <math.h>
double log(double x);
double log10(double x);
```

Return log10() returns $\log_{10}x$. If $x < 0$ log10() assigns EDOM to errno.

See Also exp(), **errno.h**

Listing 11.12 For example of usage see:

“log(), log10() example” on page 60

modf

Description Separate integer and fractional parts.

Prototype

```
#include <math.h>
double modf(double value, double *iptr);
```

Remarks The `modf()` function separates `value` into its integer and fractional parts. In other words, `modf()` separates `value` such that `value` = $f + i$ where $0 \leq f < 1$, and i is the largest integer that is not greater than `value`.

Return `modf()` returns the signed fractional part of `value`, and stores the integer part in the integer pointed to by `iptr`.

See Also `frexp(), ldexp()`

Listing 11.13 Example of modf() usage.

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

void main(void)
{
  double i, f, value = 27.04;

  f = modf(value, &i);
  printf("The fractional part of %f is %f.\n", value, f);
  printf("The integer part of %f is %f.\n", value, i);
}
```

```
Output:
The fractional part of 27.040000 is 0.040000.
The integer part of 27.040000 is 27.000000.
```

pow

Description Calculate x^y.

Prototype

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

Return `pow()` returns x^y. The `pow()` function assigns `EDOM` to `errno` if `x` is 0.0 and `y` is less than or equal to zero or if `x` is less than zero and y is not an integer.

See Also `sqrt()`, **`errno.h`**

Listing 11.14 pow() example

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

void main(void)
{
  double x;

  printf("Powers of 2:\n");
  for (x = 1.0; x <= 10.0; x += 1.0)
    printf("2 to the %4.0f is %4.0f.\n", x, pow(2, x));
}
```

```
Output:
Powers of 2:
2 to the    1 is    2.
2 to the    2 is    4.
2 to the    3 is    8.
2 to the    4 is   16.
2 to the    5 is   32.
2 to the    6 is   64.
2 to the    7 is  128.
2 to the    8 is  256.
```

```
2 to the    9 is  512.
2 to the   10 is 1024.
```

sin

Description Compute sine.

Prototype

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

Remarks The argument for the sin() function should be in radians. One radian is equal to 360/2π degrees.

Return `sin()` returns the sine of `x`. `x` is measured in radians.

See Also `cos(), tan()`

Listing 11.15 Example of sin() usage.

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

#define DtoR 2*pi/360

void main(void)
{
  double x = 57.0;
  double xRad = x*DtoR;

  printf("The sine of %.2f degrees is %.4f.\n",x, sin(xRad));
}
```

```
Output:
The sine of 57.00 degrees is 0.8387.
```

sinh

Description Compute the hyperbolic sine.

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

Return `sinh()` returns the hyperbolic sine of `x`.

See Also `cosh(), tanh()`

Listing 11.16 sinh() example

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

void main(void)
{
  double x = 0.5;
  printf("Hyperbolic sine of %f is %f.\n", x, sinh(x));
}
```

```
Output:
Hyperbolic sine of 0.500000 is 0.521095.
```

sqrt

Description Calculate the square root.

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

Return `sqrt()` returns the square root of `x`.

See Also `pow()`

Listing 11.17 sqrt() example

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

void main(void)
{
  double x = 64.0;

  printf("The square root of %f is %f.\n", x, sqrt(x));
}
```

```
Output:
The square root of 64.000000 is 8.000000.
```

tan

Description Compute tangent.

Prototype

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

Return `tan()` returns the tangent of `x`. `x` is measured in radians.

See Also `cos(), sin()`

Listing 11.18 Example of tan() usage.

```
#include <math.h>
#include <stdio.h>
void main(void)
{
  double x = 0.5;
```

```
  printf("The tangent of %f is %f.\n", x, tan(x));

}
```

```
Output:
The tangent of 0.500000 is 0.546302.
```

tanh

Description Compute the hyperbolic tangent.

Prototype

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

Return `tanh()` returns the hyperbolic tangent of `x`.

See Also `cosh(), sinh()`

Listing 11.19 tanh() example

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

void main(void)
{
  double x = 0.5;

  printf("The hyperbolic tangent of %f is %f.\n",x, tanh(x));

}
```

```
Output:
The hyperbolic tangent of 0.500000 is 0.462117.
```

12

setjmp.h

The `setjmp.h` header file provides a means of saving and restoring a processor state.

Non-local jumps and exception handling

The `setjmp.h` header file provides a means of saving and restoring a processor state. The `setjmp.h` functions are typically used for programming error and low-level interrupt handlers.

The `setjmp()` function saves the current calling environment—the current processor state—in its `jmp_buf` argument. The `jmp_buf` type, an array, holds the processor program counter, stack pointer, and relevant data and address registers.

The `longjmp()` function restores the processor to its state at the time of the last `setjmp()` call. In other words, `longjmp()` returns program execution to the last `setjmp()` call if the `setjmp()` and `longjmp()` pair use the same `jmp_buf` variable as arguments.

Because the `jmp_buf` variable can be global, the `setjmp` and `longjmp` calls do not have to be in the same function body.

A `jmp_buf` variable must be initialized with a call to `setjmp()` before being used with `longjmp()`. Calling `longjmp()` with an uninitialized `jmp_buf` variable may crash the program. Variables assigned to registers through compiler optimization may be corrupted during execution between `setjmp()` and `longjmp()` calls. This situation can be avoided by declaring affected variables as `volatile`.

longjmp

Description Restore the processor state saved by `setjmp()`.

Prototype

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Remarks The `longjmp()` function restores the calling environment (i.e. returns program execution) to the state saved by the last called `setjmp()` to use the `env` variable. Program execution continues from the `setjmp()` function. The `val` argument is the value returned by `setjmp()` when the processor state is restored.

The `env` variable must be initialized by a previously executed `setjmp()` before being used by `longjmp()` to avoid undesired results in program execution.

See Also `setjmp()`, **`signal.h`**: `signal()`, **`stdlib.h`**: `abort()`

Listing 12.1 For example of long jmp() usageUsage

“setjmp() example” on page 71.

setjmp

Description Save the processor state for `longjmp()`.

Prototype

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Remarks The `setjmp()` function saves the calling environment—data and address registers, the stack pointer, and the program counter—in the `env` argument. The argument must be initialized by `setjmp()` before being passed as an argument to `longjmp()`.

Return When it is first called, `setjmp()` saves the processor state and returns 0. When `longjmp()` is called program execution jumps to the `setjmp()` that saved the processor state in env. When activated through a call to `longjmp()`, `setjmp()` returns `longjmp()`'s `val` argument.

See Also `longjmp()`, **`signal.h`**: `signal()`, **`stdlib.h`**: `abort()`

Listing 12.2 setjmp() example

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

// Let main() and doerr() both have
// access to global env
volatile jmp_buf env;

void doerr(void);
void main(void)
{

  int i, j, k;

  printf("Enter 3 integers that total less than 100.\n");
  printf("A zero sum will quit.\n\n");

  // If the total of entered numbers is not less than 100,
  // program execution is restarted from this point.

  if (setjmp(env) != 0)
    printf("Try again, please.\n");

  do {
    scanf("%d %d %d", &i, &j, &k);
    if ( (i + j + k) == 0)
      exit(0);// quit program
    printf("%d + %d + %d = %d\n\n", i, j, k, i+j+k);
    if ( (i + j + k) >= 100)
```

```
      doerr(); // error!
  } while (1);// loop forever
}

void doerr(void)// this is the error handler
{
  printf("The total is >= 100!\n");
  longjmp(env, 1);
}
```

```
Output:

Enter 3 integers that total less than 100.
A zero sum will quit.

10 20 30
10 + 20 + 30 = 60

-4 5 1000
-4 + 5 + 1000 = 1001

The total is >= 100!
Try again, please.
0 0 0
```

13

signal.h

The include file signal.h list the software interrupt specifications.

Signal handling

Signals are software interrupts. There are signals for aborting a program, floating point exceptions, illegal instruction traps, user-signaled interrupts, segment violation, and program termination. These signals, described in “signal.h Signal descriptions” on page 74, are defined as macros in the `signal.h` file.

The `signal()` function specifies how a signal is handled: a signal can be ignored , handled in a default manner, or be handled by a programmer-supplied signal handling function. “Signal handling functions” on page 75 describes the pre-defined signal handling macros that expand to functions.

Signals are invoked, or raised, using the `raise()` function. When a signal is raised its associated function is executed.

With the Metrowerks C implementation of `signal.h` a signal can only be invoked through the `raise()` function, and, in the case of the `SIGABRT` signal, through the `abort()` function. When a signal is raised, its signal handling function is executed as a normal function call.

The default signal handler for all signals except `SIGTERM` is `SIG_DFL`. The `SIG_DFL` function aborts a program with the `abort()` function, while the `SIGTERM` signal terminates a program normally with the `exit()` function.

The ANSI C Standard Library specifies that the SIG prefix used by the signal.h macros is reserved for future use. The programmer

should avoid using the prefix to prevent conflicts with future specifications of the Standard Library.

The type typedef `char sig_atomic_t` in `signal.h` can be accessed as an incorruptible, atomic entity during an asynchronous interrupt.

Table 13.1 signal.h Signal descriptions

Macro	Description
`SIGABRT`	Abort signal. This macro is defined as a positive integer value. This signal is called by the `abort()` function.
`SIGFPE`	Floating point exception signal. This macro is defined as a positive integer value.
`SIGILL`	Illegal instruction signal. This macro is defined as a positive integer value.
`SIGINT`	Interactive user interrupt signal. This macro is defined as a positive integer value.
`SIGSEGV`	Segment violation signal. This macro is defined as a positive integer value.
`SIGTERM`	Terminal signal. This macro is defined as a positive integer value. When raised this signal terminates the calling program by calling the exit() function.

The `signal()` function specifies how a signal is handled: a signal can be ignored , handled in a default manner, or be handled by a programmer-supplied signal handling function. “Signal handling functions” on page 75 describes the pre-defined signal handling macros that expand to functions

Table 13.2 Signal handling functions

Macro	Description
`SIG_IGN`	This macro expands to a pointer to a function that returns void. It is used as a function argument in `signal()` to designate that a signal be ignored.
`SIG_DFL`	This macro expands to a pointer to a function that returns void. This signal handler quits the program without flushing and closing open streams.
`SIG_ERR`	A macro defined like `SIG_IGN` and `SIG_DFL` as a function pointer. This value is returned when signal() cannot honor a request passed to it.

signal

Description

Set signal handling

Prototype

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Remarks

The `signal()` function returns a pointer to a signal handling routine that takes an `int` value argument.

The sig argument is the signal number associated with the signal handling function. The signals defined in `signal.h` are listed in "signal.h Signal descriptions" on page 74.

The func argument is the signal handling function. This function is either programmer-supplied or one of the pre-defined signal handlers described in "Signal handling functions" on page 75.

When it is raised, a signal handler's execution is preceded by the invocation of `signal(sig, SIG_DFL)`. This call to `signal()` effectively disables the user's handler. It can be reinstalled by placing a call within the user handler to `signal()` with the user's handler as its function argument.

Return `signal()` returns a pointer to the signal handling function set by the last call to `signal()` for signal sig. If the request cannot be honored, `signal()` returns `SIG_ERR`.

See Also `raise()`, **`stdlib.h`**: `abort()`, `atexit()`, `exit()`

Listing 13.1 Example of `signal()` usage

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

void userhandler(int);

void userhandler(int sig)
{
  char c;

  printf("userhandler!\nPress return.\n");

  /* wait for the return key to be pressed */
  c = getchar();
}
void main(void)
{
  void (*handlerptr)(int);
  int i;

  handlerptr = signal(SIGINT, userhandler);
  if (handlerptr  == SIG_ERR)
    printf("Can't assign signal handler.\n");

  for (i = 0; i < 10; i++) {
    printf("%d\n", i);
    if (i == 5) raise(SIGINT);
  }
}
```

```
Output:

0
1
2
3
4
5
userhandler!
Press return.

6
7
8
9
```

raise

Description Raise a signal.

Prototype

```
#include <signal.h>
int raise(int sig);
```

Remarks The `raise()` function calls the signal handling function associated with signal sig.

Return `raise()` returns a zero if the signal is successful; it returns a non-zero value if it is unsuccessful.

See Also **`setjmp.h:`** `longjmp()`**, `signal.h:` `raise()`, `stdlib.h:`** `abort()`, `atexit()`, `exit()`

Listing 13.2 For example of rais() usage

```
Refer to the example for “Example of signal() usage” on page 76
```

14

SIOUX.h

The SIOUX library handles all the Macintosh menus, windows, and events so your program doesn't need to.

Using SIOUX

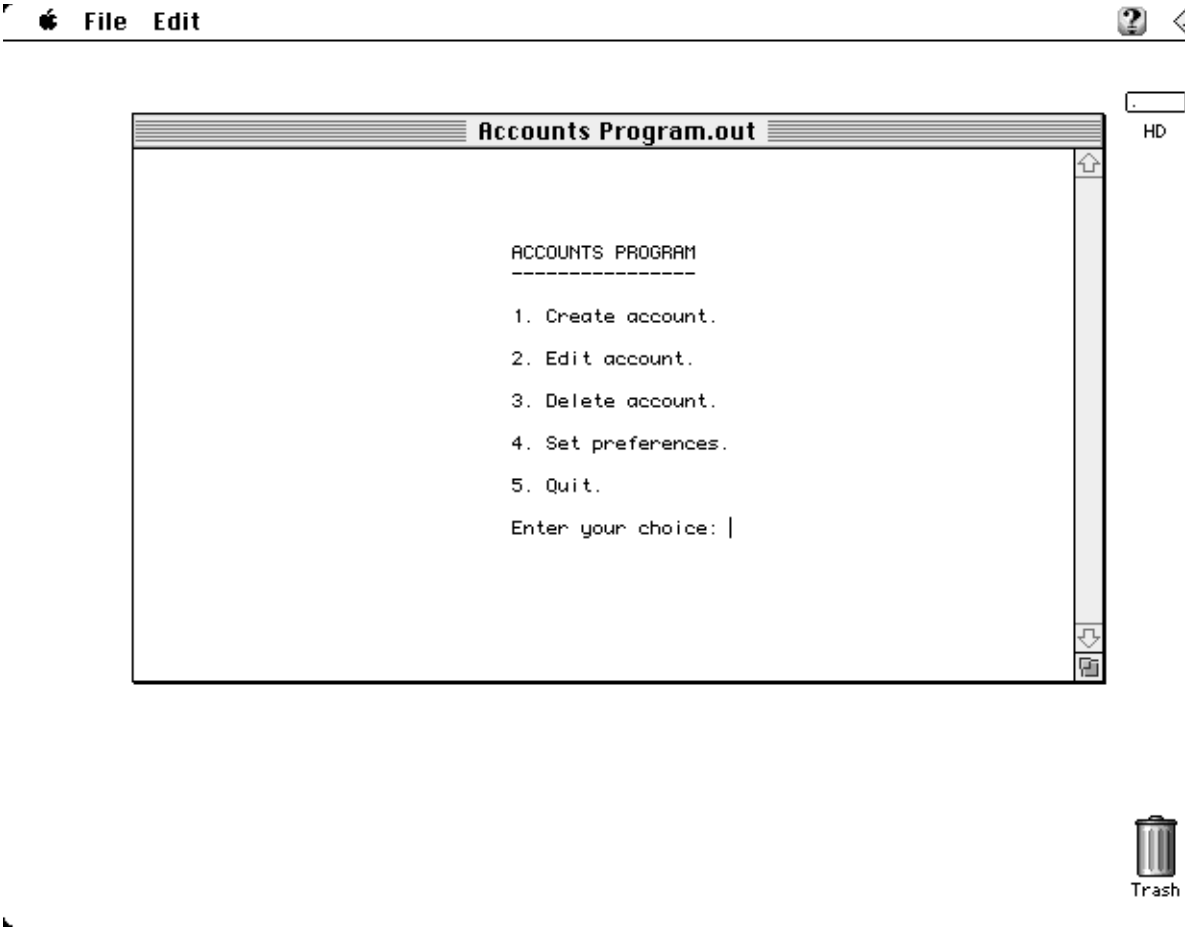
Sometimes you need to port a program that was originally written for DOS or UNIX. Or you need to write a new program quickly and don't have the time to write a complete Macintosh program that handles windows, menus, and events.

To help you, Metrowerks provides you with the SIOUX library, which handles all the Macintosh menus, windows, and events so your program doesn't need to. It creates a window that's much like a dumb terminal or TTY. You can write to it and read from it with the standard C functions and C++ operators, such as `printf()`, `scanf()`, `getchar()`, `putchar()`, <<, and >>. The SIOUX library also creates a File menu that lets you save and print the contents of the window, and an Edit menu that lets you cut, copy, and paste the contents in the window.

NOTE: If you're porting a UNIX or DOS program, you might also need the functions in `console.h` and `unix.h`.

"A Running SIOUX Program" on page 80 shows a running SIOUX program.

Figure 14.1 A Running SIOUX Program



The window is a resizable, scrolling text window, where your program reads and writes text. It saves up to 32K of your program's text.

With the commands from the Edit menu, you can cut and copy text from the SIOUX window and paste text from other applications into the SIOUX window. With the commands in the File menu, you can print or save the contents of the SIOUX window.

To stop your program at any time, press Command-Period or Control-C. The SIOUX application keeps running so you can edit or save the window's contents. If you want to exit when your program is

done or avoid the dialog asking whether to save the window, see "Changing what happens on quit" on page 88

To quit out of the SIOUX application at any time, choose Quit from the File menu. If you haven't saved the contents of the window, the application displays a dialog asking you whether you want to save the contents of the window now. If you want to remove the status line, see "Showing the status line" on page 89.

Creating a Project with SIOUX

To use the SIOUX library, create a project from a project stationery pads that creates an ANSI project, but not a Strict ANSI project. For example, you can create one from ~ANSI 68K (2i) C++.µ or ~ANSI PPC C .µ but not ~Strict ANSI 68K (2i) C++.µ.

If you want only to write to or read from standard input and output, you don't need to call any special functions or include any special header files. When your program refers to standard input or output, the SIOUX library kicks in automatically and creates a SIOUX window for it.

NOTE: IIn this chapter, standard input and standard output refer to `stdin`, `stdout`, `stderr`, `cin`, `cout`, and `cerr`. Remember that functions like `printf()` and `scanf()` use standard input and output even though these symbols do not appear in their parameter lists.

If you want to customize the SIOUX environment, you must #include `SIOUX.h` and modify `SIOUXSettings` before you use standard input or output. As soon as you use one of them, SIOUX creates a window and you cannot modify it. For more information, see "Customizing SIOUX" on page 82.

If you want to use a SIOUX window in a program that has its own event loop, you must modify `SIOUXSettings` and call the function `SIOUXHandleOneEvent()`. For more information, see "Using SIOUX windows in your own application" on page 89.

If you want to add SIOUX to a project you already created, the project must contain certain libraries.

A 68K project must contain at least these libraries:

- SIOUX.68K.Lib
- MacOS.Lib
- ANSI.C .Lib suitable for your 68k project version

A PPC project must contain at least these libraries:

- SIOUX.PPC.Lib
- InterfaceLib
- MWCRuntime.lib
- MathLib
- ANSI C.PPC.Lib

Customizing SIOUX

This following sections describe how you can customize the SIOUX environment by modifying the structure `SIOUXSettings`. SIOUX examines the data fields of `SIOUXSettings` to determine how to create the SIOUX window and environment.

NOTE: ITo customize SIOUX, you must modify `SIOUXSettings` before you call any function that uses standard input or output. If you modify `SIOUXSettings` afterwards, SIOUX does not change its window.

The first three sections, “Changing the font and tabs” on page 85, “Changing the size and location” on page 87, and “Showing the status line” on page 89, describe how to customize the SIOUX window. The next section, “Changing what happens on quit” on page 88, describe how to modify how SIOUX acts when you quit it. The last section, “Using SIOUX windows in your own application” on page 89, describes how you can use a SIOUX window in your own Macintosh program.

"The SIOUXSettings structure" on page 83 summarizes what's in the `SIOUXSettings` structure.

Table 14.1 The SIOUXSettings structure

This field…		Specifies…
char	initializeTB	Whether to initialize the Macintosh toolbox.
char	standalone	Whether to use your own event loop or SIOUX's.
char	setupmenus	Whether to create File and Edit menus for the application.
char	autocloseonquit	Whether to quit the application automatically when your program is done.
char	asktosaveonclose	Query the user whether to save the SIOUX output as a file, when the program is done.
char	showstatusline	Whether to draw the status line in the SIOUX window.
short	tabspaces	If greater than zero, substitute a tab with that number of spaces. If zero, print the tabs.
short	column	The number of characters per line that the SIOUX window will contain.
short	rows	The number of lines of text that the SIOUX window will contain.
short	toppixel	The location of the top of the SIOUX window.
short	leftpixel	The location of the left of the SIOUX window.

Table 14.1 The SIOUXSettings structure (continued)

This field…		Specifies…
short	fontid	The font in the SIOUX window.
short	fontsize	The size of the font in the SIOUX window.
short	fontface	The style of the font in the SIOUX window.

"Example of customizing a SIOUX Window" on page 84 contains a small program that customizes a SIOUX window, and "A Customized SIOUX Window." shows what the window looks like.

Listing 14.1 Example of customizing a SIOUX Window

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

void main(void)
{
  /*
Don't exit the program after it runs or ask whether to save the
window when the program exit  */
  SIOUXSettings.autocloseonquit = FALSE;
  SIOUXSettings.asktosaveonclose = FALSE;

  /* Don't show the status line */
  SIOUXSettings.showstatusline = FALSE;

  /* Make the window large enough to fit 1 line of text that
contains 12 characters. */

  SIOUXSettings.columns = 12;
  SIOUXSettings.rows = 1;

  /* Place the window's top left corner at (5,40).  */
  SIOUXSettings.toppixel = 40;
```

```
  SIOUXSettings.leftpixel = 5;

  /* Set the font to be 48-point, bold, italic Times.   */
  SIOUXSettings.fontsize = 48;
  SIOUXSettings.fontface = bold + italic;
  SIOUXSettings.fontid = times;

  printf("Hello World!");
}
```

Figure 14.2 A Customized SIOUX Window

Changing the font and tabs

This section describes how to change how SIOUX handles tabs with the field `tabspaces` and how to change the font with the fields `fontid`, `fontsize`, and `fontface`.

NOTE: IThe status line in the SIOUX window writes its messages with the font specified in the fields `fontid`, `fontsize`, and `fontface`. If that font is too large, the status line may be unreadable. You can remove the status line by setting the field `showstatusline` to FALSE, as described in “Showing the status line” on page 89.

To change the font in the SIOUX window, set `fontid` to one of these values:

- `courier`

- `geneva`
- `helvetica`
- `monaco`
- `newYork` (note the capitalization)
- `symbol`
- `times`

By default, fontid is monaco.

To change the character style for the font, set fontface to one of these values:

- `normal`
- `bold`
- `italic`
- `underline`
- `outline`
- `shadow`
- `condense`
- `extend`

To combine styles, add them together. For example, to write text that's bold and italic, set `fontface` to `bold + italic`. By default, `fontface` is `normal`.

To change the size of the font, set `fontsize` to the size. By default, `fontsize` is `9`.

The field `tabspaces` controls how SIOUX handles tabs. If `tabspaces` is any number greater than `0`, SIOUX prints that number of spaces instead of a tab. If `tabspaces` is `0`, it prints a tab. In the SIOUX window, a tab looks like a single space, so if you are printing a table, you should set `tabspaces` to an appropriate number, such as 4 or 8. By default, `tabspaces` is 4.

The sample below sets the font to 12-point, bold, italic New York and substitutes 4 spaces for every tab:

```
SIOUXSettings.fontsize = 12;
SIOUXSettings.fontface = bold + italic;
SIOUXSettings.fontid = newYork;
SIOUXSettings.tabspaces = 4;
```

Changing the size and location

SIOUX lets you change the size and location of the SIOUX window.

To change the size of the window, set `rows` to the number of lines of text in the window and set `columns` to the number of characters in each line. SIOUX checks the font you specified in `fontid`, `fontsize`, and `fontface` and creates a window that will be large enough to contain the number of lines and characters you specified. If the window is too large to fit on your monitor, SIOUX creates a window only as large as the monitor can contain.

For example, the code below creates a window that contains 10 lines with 40 characters per line:

```
SIOUXSettings.rows = 10;
SIOUXSettings.columns = 40;
```

By default, the SIOUX window contains 24 rows with 80 characters per row.

To change the position of the SIOUX window, set `toppixel` and `leftpixel` to the point where you want the top left corner of the SIOUX window to be. By setting `toppixel` to `38` and `leftpixel` to 0, you can place the window as far left as possible and just under the menu bar. Notice that if `toppixel` is less than `38`, the SIOUX window is under the menu bar. If `toppixel` and `leftpixel` are both 0, SIOUX doesn't place the window at that point but instead centers it on the monitor.

For example, the code below places the window just under the menu bar and near the left edge of the monitor:

```
SIOUXSettings.toppixel = 40;
SIOUXSettings.leftpixel = 5;
```

Changing what happens on quit

The fields `autocloseonquit` and `asktosaveonclose` let you control what SIOUX does when your program is over and SIOUX closes its window.

The field `autocloseonquit` determines what SIOUX does when your program has finished running. If `autocloseonquit` is `TRUE`, SIOUX automatically exits. If `autocloseonquit` is `FALSE`, SIOUX continues to run, and you must choose Quit from the File menu to exit. By default, `autocloseonquit` is `FALSE`.

TIP: **T**You can save the contents of the SIOUX window at any time by choosing Save from the File menu.

The field `asktosaveonclose` determines what SIOUX does when it exits. If `asktosaveonclose` is `TRUE`, SIOUX displays a dialog asking whether you want to save the contents of the SIOUX window. If `asktosaveonclose` is `FALSE`, SIOUX exits without displaying the dialog. By default, `asktosaveonclose` is `TRUE`.

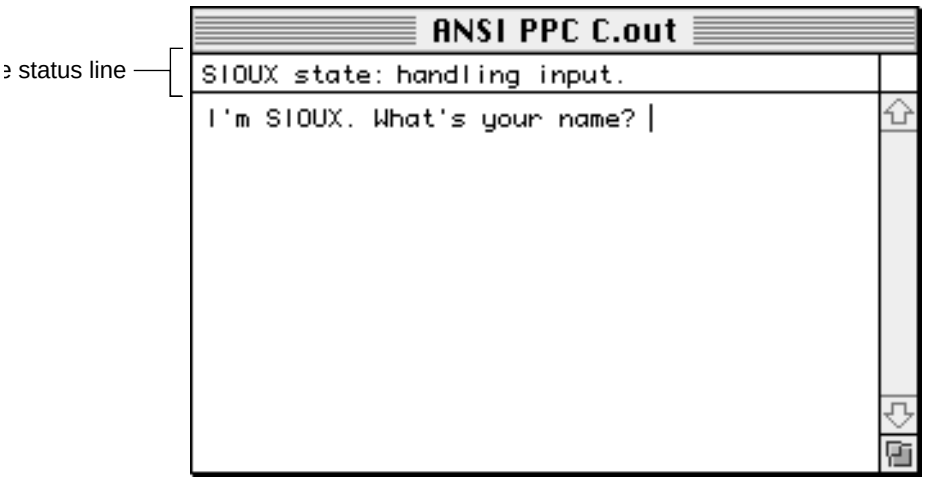
For example, the code below quits the SIOUX application as soon as your program is done and doesn't ask you to save the output:

```
SIOUXSettings.autocloseonquit = TRUE;
SIOUXSettings.asktosaveonclose = FALSE;
```

Showing the status line

The field `showstatusline` lets you control whether the SIOUX window displays a status line, which contains such information as whether the program is running, handling output, or waiting for input. If `showstatusline` is `TRUE`, the status line is displayed. If `showstatusline` is `FALSE`, the status line is not displayed. By default, `showstatusline` is `FALSE`.

Figure 14.3 The Status Line



Using SIOUX windows in your own application

This section explains how you can limit how much SIOUX controls your program. But first, you need to understand how SIOUX works with your program. You can consider the SIOUX environment to be an application that calls your `main()` function as just another function. Before SIOUX calls `main()`, it performs some initialization to set up the Macintosh Toolbox and its menu. After `main()` completes, SIOUX cleans up what it created. Even while `main()` is running, SIOUX sneaks in whenever it performs input or output, acting on any menu you've chosen or command key you've pressed.

However, SIOUX lets you choose how much work it does for you. You can choose to handle your own events, set up your own menus, and initialize the Macintosh Toolbox yourself.

When you want to write an application that handles its own events and uses SIOUX windows for easy input and output, set the field `standalone` to `FALSE` before you use standard input or output. SIOUX doesn't use its event loop and sets the field `autocloseonquite` to `TRUE` for you, so the application exits as soon as your program is done. In your event loop, you need to call the function `SIOUXHandleOneEvent()`, described on "SIOUXHandleOneEvent" on page 90.

When you don't want to use SIOUX's menus, set the field `setupmenus` to `FALSE`. If `standalone` is also `FALSE`, you won't be able to create menus, and your program will have none. If `standalone` is `TRUE`, you can create and handle your own menus.

When you want to initialize the Macintosh Toolbox yourself, set the field `initializeTB` to `FALSE`. The field `standalone` does not affect `initializeTB`.

For example, these lines set up SIOUX for an application that handles its own events, creates its own menus, and initializes the Toolbox:

```
SIOUXSettings.standalone = FALSE;
SIOUXSettings.setupmenus = FALSE;
SIOUXSettings.initializeTB = FALSE;
```

SIOUXHandleOneEvent

Description Handles an event for a SIOUX window.

Prototype

```
#include <sioux.h>
Boolean SIOUXHandleOneEvent (EventRecord *event);
```

Remarks Before you handle an event, call `SIOUXHandleOneEvent()` so SIOUX can update its windows when necessary. The argument `event` should be an event that `WaitNextEvent()` or `GetNextEvent()` returned. The function returns `TRUE` if it handled the

event and FALSE if it didn't. If event is a NULL pointer, the function polls the event queue until it receives an event.

Return If it handles the event, SIOUXHandleOneEvent() returns TRUE. Otherwise, SIOUXHandleOneEvent() returns FALSE.

Listing 14.2 Example of SIOUXHandleOneEvent() usage.

```
void MyEventLoop(void)
{
  EventRecord event;
  RgnHandle cursorRgn;
  Boolean gotEvent, SIOUXDidEvent;

  cursorRgn = NewRgn();

  do {
    gotEvent = WaitNextEvent(everyEvent, &event,
            MyGetSleep(), cursorRgn);

    /* Before handling the event on your own,
     * call SIOUXHandleOneEvent() to see whether
     * the event is for SIOUX.
     */
    if (gotEvent)
      SIOUXDidEvent = SIOUXHandleOneEvent(&event);

    if (!SIOUXDidEvent)
      DoEvent(&event);

  } while (!gDone)
}
```

SIOUXSetTitle

Description To set the title of the SIOUX output window.

Prototype

```
include <SIOUX.h>
extern void SIOUXSetTitle
  (unsigned char title[256])
```

Remarks You must call the `SIOUXSetTitle()` function after an output to the `SIOUX` window. The function `SIOUXSetTitle()` does not return an error if the title is not set. A write to console is not performed until a new line is written, the stream is flushed or the end of the program occurs.

WARNING! TThe argument for `SIOUXSetTitle()` is a pascal string, not a C style string.

Return There is no return value from `SIOUXSetTitle()`

Listing 14.3 Example of `SIOUXSetTitle()` usage.

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

void main(void)
{
  printf("Hello World\n");
  SIOUXSetTitle("\pMy Title");
}
```

15

stat.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the Macintosh Toolbox.

NOTE: If you're porting a UNIX or DOS program, you might also need the functions in `console.h` and `SIOUX.h`.

fstat

Purpose Gets information about an open file.

Prototype

```
#include <stat.h>
int fstat(int fildes, struct stat *buf);
```

Remarks This function gets information on the file associated with `fildes` and puts the information in the structure that `buf` points to. The structure contains the fields listed in "The stat structure" on page 94.

Table 15.1 The stat structure

This field		is the
mode_t	st_mode	File type. It can be one of the following: `S_IFDIR`, if this is a directory. `S_IFLNK`, if this is an alias. `S_IFREG`, if this is a file.
ino_t	st_ino	File ID for this file.
dev_t	st_dev	Volume reference number of the device that contains this file.
nlink_t	st_nlink	Number of links . This is 2 for a directory or 1 for a file.
uid_t	st_uid	User ID of the file's owner. Since the Macintosh does not have anything similar, this is a simulated value that a typical user process running under UNIX might have.
gid_t	st_gid	Group ID of the file's group. Since the Macintosh does not have anything similar, this is a simulated value that a typical user process running under UNIX might have.
dev_t	st_redev	Device type. Always 0 in Metrowerks C/C++.
off_t	st_size	File size in bytes.
time_t	st_atime	Time the file was last modified. This field always has the same value as `st_mtime`.
time_t	st_mtime	Time the file was last modified. This field always has the same value as `st_atime`.
time_t	st_ctime	Time the file was created.
long	st_blksize	The size each block on the device that contains this file.
long	st_blocks	The number of blocks this file contains.

Return If it is successful, `fstat()` returns zero. If it encounters an error, `fstat()` returns `-1` and sets `errno`.

See Also **`unix.h`**: `stat(), uname()`

Listing 15.1 Example of fstat() usage.

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

void main(void)
{
  struct stat info;
  int fd;

  fd = open("mytest", O_WRONLY | O_CREAT | O_TRUNC);
  write(fd, "Hello world!\n", 13);

  fstat(fd, &info);
    /* Get information on the open file.              */

  printf("File mode:         0x%lX\n", info.st_mode);
  printf("File ID:           0x%lX\n", info.st_ino);
  printf("Volume ref. no.:   0x%lX\n", info.st_dev);
  printf("Number of links:   %hd\n", info.st_nlink);
  printf("User ID:           %lu\n", info.st_uid);
  printf("Group ID:          %lu\n", info.st_gid);
  printf("Device type:       %d\n", info.st_rdev);
  printf("File size:         %ld\n", info.st_size);
  printf("Access time:       %s", ctime(&info.st_atime));
  printf("Modification time: %s", ctime(&info.st_mtime));
  printf("Creation time:     %s", ctime(&info.st_ctime));
  printf("Block size:        %ld\n", info.st_blksize);
  printf("Number of blocks:  %ld\n", info.st_blocks);

  close(fd);
}
```

```
This program may print the following:
File mode:         0x800
File ID:           0x5ACA
```

```
Volume ref. no.:   0xFFFFFFFF
Number of links:   1
User ID:           200
Group ID:          100
Device type:       0
File size:         13
```

mkdir

Purpose Makes a folder.

Prototype

```
#include <stat.h>
int mkdir(const char *path, int mode);
```

Remarks This function creates the new folder specified in `path`. It ignores the argument `mode`.

Return If it is successful, `mkdir()` returns zero. If it encounters an error, `mkdir()` returns `-1` and sets `errno`.

See Also **`unix.h`**: `unlink(), rmdir()`

Listing 15.2 Example for mkdir()

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

void main(void)
{
  mkdir("Akbar:test ƒ", 0);
}
```

```
Creates a folder named test ƒ on the volume Akbar
```

stat

Purpose Gets information about a file.

Prototype

```
#include <stat.h>
int stat(const char *path, struct stat *buf);
```

Remarks This function gets information on the file specified in `path` and puts the information in the structure that `buf` points to. The structure contains the fields listed in "The stat structure" on page 97.

Table 15.2 The stat structure

This field…		is the…
mode_t	st_mode	File type. It can be one of the following: `S_IFDIR`, if this is a directory. `S_IFLNK`, if this is an alias. `S_IFREG`, if this is a file.
ino_t	st_ino	File ID for this file.
dev_t	st_dev	Volume reference number of the device that contains this file.
nlink_t	st_nlink	Number of links . This is 2 for a directory or 1 for a file.
uid_t	st_uid	User ID of the file's owner. Since the Macintosh does not have anything similar, this is a simulated value that a typical user process running under UNIX might have.
gid_t	st_gid	Group ID of the file's group. Since the Macintosh does not have anything similar, this is a simulated value that a typical user process running under UNIX might have.
dev_t	st_redev	Device type. Always 0 in Metrowerks C/C++.
off_t	st_size	File size in bytes.
time_t	st_atime	Time the file was last modified. This field always has the same value as `st_mtime`.

Table 15.2 The stat structure

This field…		is the…
time_t	st_mtime	Time the file was last modified. This field always has the same value as `st_atime`.
time_t	st_ctime	Time the file was created.
long	st_blksize	The size each block on the device that contains this file.
long	st_blocks	The number of blocks this file contains.

Return If it is successful, `stat()` returns zero.

See Also **`unix.h`:** `fstat()`, `uname()`

Listing 15.3 Example of stat() usage.

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

void main(void)
{
  struct stat info;

  stat("Akbar:System Folder:System", &info);
    /* Get information on the System file.              */

  printf("File mode:         0x%lX\n", info.st_mode);
  printf("File ID:           0x%lX\n", info.st_ino);
  printf("Volume ref. no.:   0x%lX\n", info.st_dev);
  printf("Number of links:   %hd\n", info.st_nlink);
  printf("User ID:           %lu\n", info.st_uid);
  printf("Group ID:          %lu\n", info.st_gid);
  printf("Device type:       %d\n", info.st_rdev);
  printf("File size:         %ld\n", info.st_size);
  printf("Access time:       %s", ctime(&info.st_atime));
```

```
  printf("Modification time: %s", ctime(&info.st_mtime));
  printf("Creation time:     %s", ctime(&info.st_ctime));
  printf("Block size:        %ld\n", info.st_blksize);
  printf("Number of blocks:  %ld\n", info.st_blocks);
}
```

```
This program may print the following:
File mode:         0x800
File ID:           0x4574
Volume ref. no.:   0x0
Number of links:   1
User ID:           200
Group ID:          100
Device type:       0
File size:         30480
Access time:       Mon Apr 17 19:46:37 1995
Modification time: Mon Apr 17 19:46:37 1995
Creation time:     Fri Oct  7 12:00:00 1994
Block size:        11264
Number of blocks:  3
```

16

stdarg.h

The `stdarg.h` header file allows the creation of functions that accept a variable number of arguments.

Variable arguments for functions

The `stdarg.h` header file allows the creation of functions that accept a variable number of arguments.

A variable-length argument function is defined with an ellipsis (`...`) as its last argument. For example:

```
int funnyfunc(int a, char c, ...);
```

The function is written using the `va_list` type, the `va_start()`, `va_arg()` and the `va_end()` macros.

The function has a `va_list` variable declared within it to hold the list of function arguments. The `va_start()` macro initializes the `va_list` variable and is called before gaining access to the arguments. The `va_arg()` macro returns each of the arguments in `va_list`. When all the arguments have been processed through `va_arg()`, the `va_end()` macro is called to allow a normal return from the function.

va_arg

Description Macro to return an argument value.

Prototype

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Remarks The `va_arg()` macro returns the next argument on the function's argument list. The argument returned has the type defined by *type*. The `ap` argument must first be initialized by the `va_start()` macro.

Return The `va_arg()` macro returns the next argument on the function's argument list of type *type*.

See Also **`stdarg.h`**: `va_end(), va_start()`

Listing 16.1 For example of va() usage

```
Refer to the example “Example of va_start() usage.” on page 103.
```

va_end

Description Prepare a normal function return.

Prototype

```
#include <stdarg.h>
void va_end(va_list ap);
```

Remarks The `va_end()` function cleans the stack to allow a proper function return. The function is called after the function's arguments are accessed with the `va_arg()` macro.

See Also **`stdarg.h`**: `va_arg(), va_start()`

Listing 16.2 For example of va_end usage

```
Refer to the example “Example of va_start() usage.” on page 103.
```

va_start

Description Initialize the variable-length argument list.

Prototype

```
#include <stdarg.h>
void va_start(va_list ap, ParmN);
```

Remarks The `va_start()` macro initializes and assigns the argument list to `ap`. The *ParmN* parameter is the last named parameter before the ellipsis (`...`) in the function prototype.

See Also **`stdarg.h`**: `va_arg()`, `va_end()`

Listing 16.3 Example of va_start() usage.

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

void multisum(int *dest, ...);

void main(void)
{
  int all;

  all = 0;
  multisum(&all, 13, 1, 18, 3, 0);
  printf("%d\n", all);
}

void multisum(int *dest, ...)
{
  va_list ap;
  int n, sum = 0;

  va_start(ap, dest);

  while ((n = va_arg(ap, int)) != 0)
```

```
    sum += n; /* add next argument to dest */
  *dest = sum;
  va_end(ap); /* clean things up before leaving */
}
```

```
 Output:
35
```

17

stddef.h

The `stddef.h` header file defines commonly used macros and types that are used throughout the ANSI C Standard Library.

Commonly used definitions

The `stddef.h` header file defines commonly used macros and types that are used throughout the ANSI C Standard Library.

The `NULL` macro is the null pointer constant used in the Standard Library.

The `offsetof(structure, member)` macro expands to an integral expression of type `size_t`. The value returned is the offset in bytes of a member, `member`, from the base of its structure, `structure`.

The `ptrdiff_t` type is the signed integral type used for subtracting one pointer's value from another.

The `size_t` type is an unsigned integral type returned by the `sizeof()` operator.

The `wchar_t` type is an integral type capable of holding all character representations of the ASCII character set. In reality, `wchar_t` is defined as

typedef char wchar_t;.

18

stdio.h

The `stdio.h` header file provides functions for input/output control.

Standard input/output

The `stdio.h` header file provides functions for input/output control. There are functions for creating, deleting, and renaming files, functions to allow random access, as well as to write and read text and binary data.

Streams

A stream is an abstraction of a file designed to reduce hardware I/O requests. Without buffering, data on an I/O device must be accessed one item at a time. This inefficient I/O processing slows program execution considerably. The `stdio.h` functions use buffers in primary storage to intercept and collect data as it is written to or read from a file. When a buffer is full its contents are actually written to or read from the file, thereby reducing the number of I/O accesses. A buffer's contents can be sent to the file prematurely by using the `fflush()` function.

The `stdio.h` header offers three buffering schemes: unbuffered, block buffered, and line buffered. The `setvbuf()` function is used to change the buffering scheme of any output stream.

When an output stream is unbuffered, data sent to it are immediately read from or written to the file.

When an output stream is block buffered, data are accumulated in a buffer in primary storage. When full, the buffer's contents are sent to the destination file, the buffer is cleared, and the process is repeated

until the stream is closed. Output streams are block buffered by default if the output refers to a file.

A line buffered output stream operates similarly to a block buffered output stream. Data are collected in the buffer, but are sent to the file when the line is completed with a newline character (`'\n'`).

A stream is declared using a pointer to a `FILE`. There are three `FILE` pointers that are automatically opened for a program: `FILE *stdin`, `FILE *stdout`, and `FILE *stderr`. The `FILE` pointers `stdin` and `stdout` are the standard input and output files, respectively, for interactive console I/O. The `stderr` file pointer is the standard error output file, where error messages are written to. The `stderr` stream is written to the console. The `stdin`, `stdout`, `stderr` streams are line buffered.

For more information on routing `stdin`, `stdout`, and `stderr` to a Macintosh console window, see the chapter on SIOUX.h

File position indicator

The file position indicator is another concept introduced by the `stdio.h` header. Each opened stream has a file position indicator acting as a cursor within a file. The file position indicator marks the character position of the next read or write operation. A read or write operation advances the file position indicator. Other functions are available to adjust the indicator without reading or writing, thus providing random access to a file.

Note that console streams, `stdin`, `stdout`, and `stderr` in particular, do not have file position indicators.

End-of-file and errors

Many functions that read from a stream return the `EOF` value, defined in `stdio.h`. The `EOF` value is a nonzero value indicating that the end-of-file has been reached during the last read or write.

Some `stdio.h` functions also use the `errno` global variable. Refer to the `errno.h` header section. The use of `errno` is described in the relevant function descriptions below.

clearerr

Description Clear a stream's end-of-file and error status.

Prototype

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

Remarks The `clearerr()` function resets the end-of-file status and error status for `stream`. The end-of-file status and error status are also reset when a stream is opened.

See Also **`stdio.h`**: `feof()`, `ferror()`, `fopen()`, `fseek()`, `rewind()`

Listing 18.1 Example of clearerr() usage.

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

void main(void)
{
  FILE *f;

  static char name[] = "myfoo";
  char buf[80];

  // create a file for output
  if ( (f = fopen(name, "w")) == NULL) {
    printf("Can't open %s.\n", name);
    exit(1);
  }
  // output text to the file
  fprintf(f, "chair table chest\n");
  fprintf(f, "desk raccoon\n");
```

```
  // close the file
  fclose(f);

  // open the same file again for input
  if ( (f = fopen(name, "r")) == NULL) {
    printf("Can't open %s.\n", name);
    exit(1);
  }


  // read all the text until end-of-file
  for (; feof(f) == 0; fgets(buf, 80, f))
    fputs(buf, stdout);

  printf("feof() for file %s is %d.\n", name, feof(f));
  printf("Clearing end-of-file status. . .\n");
  clearerr(f);
  printf("feof() for file %s is %d.\n", name, feof(f));

  // close the file
  fclose(f);
}
```

```
Output
chair table chest
desk raccoon
feof() for file myfoo is 256.
Clearing end-of-file status. . .
feof() for file myfoo is 0.
```

fclose

Description Close an open file.

Prototype

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

Remarks The `fclose()` function closes a file created by `fopen()`, `freopen()`, or `tmpfile()`. The function flushes any buffered data to its file and closes the stream. After calling `fclose()`, `stream` is no longer valid and cannot be used with file functions unless it is reassigned using `fopen()`, `freopen()`, or `tmpfile()`.

All of a program's open streams are flushed and closed when a program terminates normally.

`fclose()` closes then deletes a file created by `tmpfile()`.

Return `fclose()` returns a zero if it is successful and returns a -1 if it fails to close a file.

See Also **`stdio.h`**: `fopen()`, `freopen()`, `tmpfile()`, **`stdlib.h`**: `exit()`, `abort()`

Listing 18.2 Example of fclose() usage.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  FILE *f;
  static char name[] = "myfoo";

  // create a new file for output
  if ( (f = fopen(name, "w")) == NULL) {
    printf("Can't open %s.\n", name);
    exit(1);
  }
  // output text to the file
  fprintf(f, "pizza sushi falafel\n");
  fprintf(f, "escargot sprocket\n");

  // close the file
  if (fclose(f) == -1) {
    printf("Can't close %s.\n", name);
    exit(1);
```

```
  }
}
```

feof

Description Check the end-of-file status of a stream.

Prototype

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

Remarks The `feof()` function checks the end-of-file status of the last read operation on `stream`. The function does not reset the end-of-file status.

Return `feof()` returns a nonzero value if the stream is at the end-of-file and return zero if the stream is not at the end-of-file.

See Also **`stdio.h`**: `clearerr()`, `ferror()`

Listing 18.3 Example of feof() usage.

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

void main(void)
{
  FILE *f;
  static char filename[80], buf[80] = "";

  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);

  // open the file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
```

```
    exit(1);
  }

  // read text lines from the file until
  // feof() indicates the end-of-file
  for (; feof(f) == 0 ; fgets(buf, 80, f) )
    printf(buf);

  // close the file
  fclose(f);

}
```

```
Output:
Enter a filename to read.
itwerks
The quick brown fox
jumped over the moon.
```

ferror

Description Check the error status of a stream.

Prototype

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

Remarks The `ferror()` function returns the error status of the last read or write operation on `stream`. The function does not reset its error status.

Return `ferror()` returns a nonzero value if `stream`'s error status is on, and returns zero if `stream`'s error status is off.

See Also **`stdio.h`**: `clearerr(), feof()`

Listing 18.4 Example of ferror() usage.

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

void main(void)
{
  FILE *f;
  char filename[80], buf[80];
  int ln = 0;

  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);

  // open the file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }

  // read the file one line at a time until end-of-file
  do {
    fgets(buf, 80, f);
    printf("Status for line %d: %d.\n", ln++, ferror(f));
  } while (feof(f) == 0);



  // close the file
  fclose(f);
}
```

```
Output:
Enter a filename to read.
itwerks
Status for line 0: 0.
```

```
Status for line 1: 0.
Status for line 2: 0.
```

fflush

Description Empty a stream's buffer to its file.

Prototype

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

Remarks The `fflush()` function empties `stream`'s buffer to the file associated with `stream`.

Return `fflush()` returns a nonzero value if it is unsuccessful and returns zero if it is successful.

See Also **`stdio.h`**: `setvbuf()`

Listing 18.5 Example of fflush() usage

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

void main(void)
{
  FILE *f;
  int count;


  // create a new file for output
  if (( f = fopen("foofoo", "w")) == NULL) {
    printf("Can't open file.\n");
    exit(1);
  }
  for (count = 0; count < 100; count++) {
    fprintf(f, "%5d\n", count);
```

```
    if (count % 10)
      fflush(f);// flush buffer every 10 numbers
  }
  fclose(f);
}
```

fgetc

Description Read the next character from a stream.

Prototype

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

Remarks The `fgetc()` function reads the next character from `stream` and advances its file position indicator.

Return `fgetc()` returns the character as an `int`. If the end-of-file has been reached, `fgetc()` returns `EOF`.

See Also **`stdio.h`**: `getc()`, `getchar()`

Listing 18.6 Example of fgetc() usage.

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

void main(void)
{
  FILE *f;
  char filename[80], c;



  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);
```

```
  // open the file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }

  // read the file one character at a time until
  // end-of-file is reached
  while ( (c = fgetc(f)) != EOF)
    putchar(c);// print the character

  // close the file
  fclose(f);
}
```

fgetpos

Description Get a stream's current file position indicator value.

Prototype

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Remarks The `fgetpos()` function is used in conjunction with the `fsetpos()` function to allow random access to a file. The `fgetpos()` function gives unreliable results when used with streams associated with a console (`stdin`, `stderr`, `stdout`).

While the `fseek()` and `ftell()` functions use `long` integers to read and set the file position indicator, `fgetpos()` and `fsetpos()` use `fpos_t` values to operate on larger files. The `fpos_t` type, defined in `stdio.h`, can hold file position indicator values that do not fit in a `long int`.

The `fgetpos()` function stores the current value of the file position indicator for `stream` in the `fpos_t` variable `pos` points to.

Return `fgetpos()` returns zero when successful and returns a nonzero value when it fails.

See Also **`stdio.h`**: `fseek()`, `fsetpos()`, `ftell()`

Listing 18.7 Example of fgetpos() usage.

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

void main(void)
{
  FILE *f;
  fpos_t pos;
  char filename[80], buf[80];

  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);

  // open the file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }
  printf("Reading each line twice.\n");

  // get the initial file position indicator value
  // (which is at the beginning of the file)
  fgetpos(f, &pos);

  // read each line until end-of-file is reached
  while (fgets(buf, 80, f) != NULL) {
    printf("Once: %s", buf);

    // move to the beginning of the line to read it again
    fsetpos(f, &pos);
    fgets(buf, 80, f);
    printf("Twice: %s", buf);
```

```
    // get the file position of the next line
    fgetpos(f, &pos);
  }


  // close the file
  fclose(f);

}
```

```
Output:
Enter a filename to read.
myfoo
Reading each line twice.
Once: chair table chest
Twice: chair table chest
Once: desk raccoon
Twice: desk raccoon*/
```

fgets

Description Read a character array from a stream.

Prototype

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

Remarks The `fgets()` function reads characters sequentially from `stream` beginning at the current file position, and assembles them into `s` as a character array. The function stops reading characters when `n` characters have been read. The `fgets()` function finishes reading prematurely if it reaches a newline (`'\n'`) character or the end-of-file.

Unlike the `gets()` function, `fgets()` appends the newline character (`'\n'`) to `s`. It also null terminates the character array.

Return `fgets()` returns a pointer to `s` if it is successful. If it reaches the end-of-file before reading any characters, `s` is untouched and `fgets()` returns a null pointer (`NULL`). If an error occurs `fgets()` returns a null pointer and the contents of `s` may be corrupted.

See Also **`stdio.h`**: `gets(), fprintf(), printf()`

Listing 18.8 For example of fgets() usage

Refer to “Example of feof() usage.” on page 112 for feof().

fopen

Description Open a file as a stream.

Prototype

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

Remarks The `fopen()` function opens a file specified by `filename`, and associates a stream with it. The `fopen()` function returns a pointer to a `FILE`. This pointer is used to refer to the file when performing I/O operations.

The `mode` argument specifies how the file is to be used. Table 7 describes the values for `mode`. A file opened with an update mode ("+") is buffered, so it cannot be written to and then read from (or vice versa) unless the read and write operations are separated by an operation that flushes the stream's buffer or the last read or write reached the end-of-file. The `fseek()`, `fsetpos()`, `rewind()`, and `fflush()` functions flush a stream's buffer.

All file modes, except the append modes (`"a"`, `"a+"`, `"ab"`, `"ab+"`) set the file position indicator to the beginning of the file. The append modes set the file position indicator to the end-of-file.

Table 18.1 Open modes for fopen()

Mode	Description
"r"	Open an existing text file for reading only.
"w"	Create a new text file for writing, or truncate an existing file.
"a"	Open an existing text file, or create a new one if it does not exist, for appending. Writing occurs at the end-of-file position.
"r+"	Update mode. Open an existing text file for reading and writing.
"w+"	Update mode. Open an existing text file, or create a new one, for writing and reading.
"a+"	Update mode. Open an existing text file or create a new one for reading and writing. Writing occurs at the end-of-file position.
"rb"	Open an existing binary file for reading only.
"wb"	Create a new binary file for writing, or truncate the file.
"ab"	Open an existing binary file, or create a new one if it does not exist, and append. Writing occurs are the end-of-file.
"r+b" or "rb+"	Update mode. Open an existing binary file for reading and writing.
`"w+b"` or `"wb+"`	Update mode. Open an existing binary file or create a new one for writing and reading.
`"a+b"` or `"ab+"`	Update mode. Open an existing binary file or create a new one for reading and writing. Writing occurs at the end-of-file position.

Return `fopen()` returns a pointer to a `FILE` if it successfully opens the specified file for the specified operation. `fopen()` returns a null pointer (`NULL`) when it is not successful.

See Also **`stdio.h`**: `fclose()`

Listing 18.9 Example of fopen() usage

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

void main(void)
{
  FILE *f;
  int count;

  // create a new file for output
  if (( f = fopen("foofoo", "w")) == NULL) {
    printf("Can't create file.\n");
    exit(1);
  }

  // output numbers 0 to 9
  for (count = 0; count < 10; count++)
    fprintf(f, "%5d", count);

  // close the file
  fclose(f);

  // open the file to append
  if (( f = fopen("foofoo", "a")) == NULL) {
    printf("Can't append to file.\n");
    exit(1);
  }

  // output numbers 10 to 19
  for (; count <20; count++)
    fprintf(f, "%5d\n", count);
```

```
  // close file
  fclose(f);
}
```

fprintf

Description Send formatted text to a stream.

Prototype

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

Remarks The `fprintf()` function writes formatted text to `stream` and advances the file position indicator. Its operation is the same as `printf()` with the addition of the `stream` argument. Refer to the description of `printf()`.

Return `fprintf()` returns the number of arguments written or a negative number if an error occurs.

See Also **`stdio.h`**: `printf()`, `sprintf()`, `vfprintf()`, `vprintf()`, `vsprintf()`

Listing 18.10 Example of fprintf() usage.

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

void main(void)
{
  FILE *f;
  static char filename[] = "myfoo";
  int a = 56;
  char c = 'M';
  double x = 483.582;

  // create a new file for output
```

```
  if (( f = fopen(filename, "w")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }

  // output formatted text to the file
  fprintf(f, "%10s %4.4f %-10d\n%10c", filename, x, a, c);

  // close the file
  fclose(f);
}
```

fputc

Description Write a character to a stream.

Prototype

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

Remarks The `fputc()` function writes character `c` to `stream` and advances `stream`'s file position indicator. Although the `c` argument is an `int`, it is converted to a `char` before being written to `stream`. `fputc()` is written as a function, not as a macro.

Return `fputc()` returns the character written if it is successful, and returns `EOF` if it fails.

See Also **`stdio.h`**: `putc()`, `putchar()`

Listing 18.11 Example of fputc() usage.

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

void main(void)
{
```

```
  FILE *f;
  int letter;

  // create a new file for output
  if (( f = fopen("foofoo", "w")) == NULL) {
    printf("Can't create file.\n");
    exit(1);
  }

  // output the alphabet to the file one letter
  // at a time
  for (letter = 'A'; letter <= 'Z'; letter++)
    fputc(letter, f);
  fclose(f);
}
```

fputs

Description Write a character array to a stream.

Prototype

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

Remarks The `fputs()` function writes the array pointed to by `s` to `stream` and advances the file position indicator. The function writes all characters in `s` up to, but not including, the terminating null character. Unlike `puts()`, `fputs()` does not terminate the output of `s` with a newline (`'\n'`).

Return `fputs()` returns a zero if successful, and returns a nonzero value when it fails.

See Also **`stdio.h`**: `puts()`

Listing 18.12 Example of fputs() usage.

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

void main(void)
{
  FILE *f;

  // create a new file for output
  if (( f = fopen("foofoo", "w")) == NULL) {
    printf("Can't create file.\n");
    exit(1);
  }

  // output character strings to the file
  fputs("undo\n", f);
  fputs("copy\n", f);
  fputs("cut\n", f);
  fputs("rickshaw\n", f);

  // close the file
  fclose(f);
}
```

fread

Description Read binary data from a stream.

Prototype

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
    FILE *stream);
```

Remarks The `fread()` function reads a block of binary or text data and updates the file position indicator. The data read from `stream` are stored in the array pointed to by `ptr`. The `size` and `nmemb` argu-

ments describe the size of each item and the number of items to read, respectively.

The `fread()` function reads `nmemb` items unless it reaches the end-of-file or a read error occurs.

Return `fread()` returns the number of items read successfully.

See Also **`stdio.h`**: `fgets()`, `fwrite()`

Listing 18.13 Example of fread() usage.

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


// define the item size in bytes
#define BUFSIZE  40

void main(void)
{
  FILE *f;
  static char s[BUFSIZE] = "The quick brown fox";
  char target[BUFSIZE];

  // create a new file for output and input
  if (( f = fopen("foo", "w+")) == NULL) {
    printf("Can't create file.\n");
    exit(1);
  }

  // output to the stream using fwrite()
  fwrite(s, sizeof(char), BUFSIZE, f);

  // move to the beginning of the file
  rewind(f);

  // now read from the stream using fread()
  fread(target, sizeof(char), BUFSIZE, f);
```

```
  // output the results to the console
  puts(s);
  puts(target);

  // close the file
  fclose(f);

}
```

```
Output:
The quick brown fox
The quick brown fox
freopen()
```

Description Re-direct a stream to another file.

Prototype

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

RemarkThe `freopen()` function changes the file `stream` is associated with to another file. The function first closes the file the stream is associated with, and opens the new file, `filename`, with the specified `mode`, using the same stream.

Return `fopen()` returns the value of `stream`, if it is successful. If `fopen()` fails it returns a null pointer (`NULL`).

See Also **`stdio.h`**: `fopen()`

Listing 18.14 Example of freopen() usage

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

```
void main(void)
{
  FILE *f;

  // re-direct output from the console to a new file
  if (( f = freopen("newstdout", "w+", stdout)) == NULL) {
    printf("Can't create new stdout file.\n");
    exit(1);
  }
  printf("If all goes well, this text should be in\n");
  printf("a text file, not on the screen via stdout.\n");
  fclose(f);
}
```

fscanf

Description Read formatted text from a stream.

Prototype

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Remarks The `fscanf()` function reads programmer-defined, formatted text from `stream`. The function operates identically to the `scanf()` function with the addition of the `stream` argument indicating the stream to read from. Refer to the `scanf()` function description.

Return `fscanf()` returns the number of items read. If there is an error in reading data that is inconsistent with the format string, `fscanf()` sets `errno` to a nonzero value. `fscanf()` returns `EOF` if it reaches the end-of-file.

See Also **`errno.h`, `stdio.h`**: `scanf()`

Listing 18.15 Example of fscanf() usage.

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

void main(void)
{
  FILE *f;
  int i;
  double x;
  char c;

  // create a new file for output and input
  if (( f = fopen("foobar", "w+")) == NULL) {
    printf("Can't create new file.\n");
    exit(1);
  }

  // output formatted text to the file
  fprintf(f, "%d\n%f\n%c\n", 45, 983.3923, 'M');

  // go to the beginning of the file
  rewind(f);

  // read from the stream using fscanf()
  fscanf(f, "%d %lf %c", &i, &x, &c);

  // close the file
  fclose(f);

  printf("The integer read is %d.\n", i);
  printf("The floating point value is %f.\n", x);
  printf("The character is %c.\n", c);

}
```

```
Output:
The integer read is 45.
```

```
The floating point value is 983.392300.
The character is M.
```

fseek

Description Move the file position indicator.

Prototype

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Remarks The `fseek()` function moves the file position indicator to allow random access to a file.

The function moves the file position indicator either absolutely or relatively. The `whence` argument can be one of three values defined in `stdio.h`: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.

The `SEEK_SET` value causes the file position indicator to be set `offset` bytes from the beginning of the file. In this case `offset` must be equal or greater than zero.

The `SEEK_CUR` value causes the file position indicator to be set `offset` bytes from its current position. The `offset` argument can be a negative or positive value.

The `SEEK_END` value causes the file position indicator to be set `offset` bytes from the end of the file. The `offset` argument must be equal or less than zero.

The `fseek()` function undoes the last `ungetc()` call and clears the end-of-file status of `stream`.

Return `fseek()` returns zero if it is successful and returns a nonzero value if it fails.

See Also **`stdio.h`**: `fgetpos()`, `fsetpos()`, `ftell()`

Listing 18.16 Example of fseek() usage.

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

void main(void)
{
  FILE *f;
  long int pos1, pos2, newpos;
  char filename[80], buf[80];

  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);

  // open a file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }

  printf("Reading last half of first line.\n");

  // get the file position indicator before and after
  // reading the first line
  pos1 = ftell(f);
  fgets(buf, 80, f);
  pos2 = ftell(f);
  printf("Whole line: %s\n", buf);

  // calculate the middle of the line
  newpos = (pos2 - pos1) / 2;

  fseek(f, newpos, SEEK_SET);
  fgets(buf, 80, f);
  printf("Last half: %s\n", buf);

  // close the file
  fclose(f);
```

```
}
```

```
Output:
Enter a filename to read.
itwerks
Reading last half of first line.
Whole line: The quick brown fox

Last half: brown fox
```

fsetpos

Description Set the file position indicator.

Prototype

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Remarks The `fsetpos()` function sets the file position indicator for `stream` using the value pointed to by `pos`. The function is used in conjunction with `fgetpos()` when dealing with files having sizes greater than what can be represented by the `long int` argument used by `fseek()`.

`fsetpos()` undoes the previous call to `ungetc()` and clears the end-of-file status.

Return `fsetpos()` returns zero if it is successful and returns a nonzero value if it fails.

See Also **`stdio.h`**: `fgetpos()`, `fseek()`, `ftell()`

Listing 18.17 For example of fsetpos() usage

```
Refer to “Example of fgetpos() usage.” on page 118 for fgetpos().
```

ftell

Description Return the current file position indicator value.

Prototype

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

Remarks The `ftell()` function returns the current value of stream's file position indicator. It is used in conjunction with `fseek()` to provide random access to a file.

The function will not work correctly when it is given a stream associated to a console file, such as `stdin`, `stdout`, or `stderr`, where a file indicator position is not applicable. Also, `ftell()` cannot handle files with sizes larger than what can be represented with a `long int`. In such a case, use the `fgetpos()` and `fsetpos()` functions.

Return `ftell()`, when successful, returns the current file position indicator value. If it fails, `ftell()` returns `-1L` and sets the global variable `errno` to a nonzero value.

See Also **`errno.h`, `stdio.h`**: `fgetpos()`

Listing 18.18 For example of ftell() usage

Refer to “Example of fseek() usage.” on page 132 for fseek().

fwrite

Description Write binary data to a stream.

Prototype

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
    size_t nmemb, FILE *stream);
```

Remarks The `fwrite()` function writes `nmemb` items of `size` bytes each to `stream`. The items are contained in the array pointed to by `ptr`. After writing the array to `stream`, `fwrite()` advances the file position indicator accordingly.

Return `fwrite()` returns the number of elements successfully written to `stream`.

See Also **`stdio.h`**: `fread()`

Listing 18.19 For example of fwrite() sage

Refer to “Example of fread() usage.” on page 127 for fread().

getc

Description Read the next character from a stream.

Prototype

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

Remarks The `getc()` function reads the next character from `stream`, advances the file position indicator, and returns the character as an `int` value. Unlike the `fgetc()` function, `getc()` is implemented as a macro.

Return `getc()` returns the next character from the stream or returns `EOF` if the end-of-file has been reached or a read error has occurred.

See Also **`stdio.h`**: `fgetc(), fputc(), getchar(), putchar()`

Listing 18.20 Example of getc() usage.

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

```
void main(void)
{
  FILE *f;
  char filename[80], c;

  // get a filename from the user
  printf("Enter a filename to read.\n");
  scanf("%s", filename);

  // open a file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }



  // read one character at a time until end-of-file
  while ( (c = getc(f)) != EOF)
    putchar(c);

  // close the file
  fclose(f);
}
```

getchar

Description Get the next character from `stdin`.

Prototype

```
#include <stdio.h>
int getchar(void);
```

Remarks The `getchar()` function reads a character from the `stdin` stream.

Return `getchar()` returns the value of the next character from `stdin` as an `int` if it is successful. `getchar()` returns `EOF` if it reaches an end-of-file or an error occurs.

See also: `stdio.h: fgetc(), getc(), putchar()`

Listing 18.21 Example of getchar() usage

```
#include <stdio.h>

void main(void)
{
  int c;

  printf("Enter characters to echo, * to quit.\n");

  // characters entered from the console are echoed
  // to it until a * character is read
  while ( (c = getchar()) != '*')
    putchar(c);

  printf("\nDone!\n");
}
```

```
Output:
Enter characters to echo, * to quit.
I’m experiencing deja-vu *
I’m experiencing deja-vu
Done!
```

gets

Description Read a character array from `stdin`.

Prototype

```
#include <stdio.h>
char *gets(char *s);
```

Remarks The `gets()` function reads characters from `stdin` and stores them sequentially in the character array pointed to by `s`. Characters are read until either a newline or an end-of-file is reached.

Unlike `fgets()`, the programmer cannot specify a limit on the number of characters to read. Also, `gets()` reads and ignores the newline character (`'\n'`) so that it can advance the file position indicator to the next line. The newline character is not stored `s`. Like `fgets()`, `gets()` terminates the character string with a null character.

If an end-of-file is reached before any characters are read, `gets()` returns a null pointer (`NULL`) without affecting the character array at `s`. If a read error occurs, the contents of `s` may be corrupted.

Return `gets()` returns `s` if it is successful and returns a null pointer if it fails.

See Also **`stdio.h`**: `fgets()`

Listing 18.22 Example of gets() usage.

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

void main(void)
{
  char buf[100];

  printf("Enter text lines to echo.\n");
  printf("Enter an empty line to quit.\n");

  // read character strings from the console
  // until an empty line is read
  while (strlen(gets(buf)) > 0)
    puts(buf);// puts() appends a newline to its output

  printf("Done!\n");
```

```
}
```

```
Output:
Enter text lines to echo.
Enter an empty line to quit.
I'm experiencing deja-vu
I'm experiencing deja-vu
Now go to work
Now go to work

Done!
```

perror

Description Output an error message to `stderr`.

Prototype

```
#include <stdio.h>
void perror(const char *s);
```

Remarks The `perror()` function outputs the character array pointed to by `s` and the value of the global variable `errno` to `stderr`.

See Also **`abort.h`**`: abort()`**, `errno.h`**

Listing 18.23 Example of perror() usage.

```
#include <stdio.h>

#define MAXLIST 10

void main(void)
{
  int i[MAXLIST], count;

  printf("Enter %d numbers.\n", MAXLIST);
  printf("Numbers less than 0 will generate an error.\n");
```

```
  // read MAXLIST integer values from the console
  for (count = 0; count < MAXLIST; count++) {
    scanf("%d", &i[count]);

    // if the value is <= 0 output an error message
    // to stderr using perror()
    if (i[count] < 0)
      perror("Invalid entry!\n");
  }
  printf("Done!\n");
}
```

printf

Description Output formatted text.

Prototype

```
#include <stdio.h>
int printf(const char *format,  ...);
```

Remarks The `printf()` function outputs formatted text. The function takes one or more arguments, the first being `format`, a character array pointer. The optional arguments following format are items (integers, characters, floating point values, etc.) that are to be converted to character strings and inserted into the output of `format` at specified points.

The `printf()` function sends its output to `stdout`.

The `format` character array contains normal text and conversion specifications. Conversion specifications must have matching arguments in the same order in which they occur in `format`.

The various elements of the format string is specified in the ANSI standards to be in this order from left to right.

- A percent sign
- Optional flags -,+,0,# or space

- Optional minimum field width specification
- Optional precision specification
- Optional size specification
- Conversion operator c,d,e,E,f,g,G,i,n,o,p,s,u,x,X or %

A conversion specification describes the format its associated argument is to be converted to. A specification starts with a percent sign (%), optional flag characters, an optional minimum width, an optional precision width, and the necessary, terminating conversion type. Doubling the percent sign (%%) results in the output of a single %.

An optional flag character modifies the formatting of the output; it can be left or right justified, and numerical values can be padded with zeroes or output in alternate forms. More than one optional flag character can be used in a conversion specification. "Format modifier types for printf()" on page 142 describes the flag characters.

The optional minimum width is a decimal digit string. If the converted value has more characters that the minimum width, it is expanded as required. If the converted value has fewer characters than the minimum width, it is, by default, right justified (padded on the left). If the - flag character is used, the converted value is left justified (padded on the right).

The optional precision width is a period character (.) followed by decimal digit string. For floating point values, the precision width specifies the number of digits to print after the decimal point. For integer values, the precision width functions identically to, and cancels, the minimum width specification. When used with a character array, the precision width indicates the maximum width of the output.

A minimum width and a precision width can also be specified with an asterisk (*) instead of a decimal digit string. An asterisk indicates that there is a matching argument, preceding the conversion argument, specifying the minimum width or precision width.

The terminating character, the conversion type, specifies the conversion applied to the conversion specification's matching argument. “Format modifier types for printf()” on page 142 describes the conversion type characters.

Table 18.2 Format modifier types for printf()

Modifier	Description
	Size
h	The h flag indicates that the corresponding argument is a `short int` or `unsigned short int`.
l	The lower case L indicates the argument is a `long int` or `unsigned long int.`
L	The upper case L indicates the argument is a `long double.`
	Flags
-	The conversion will be left justified.
+	The conversion, if numeric, will be prefixed with a sign (+ or -). By default, only negative numeric values are prefixed with a minus sign (-).
space	If the first character of the conversion is not a sign character, it is prefixed with a space. Because the plus sign flag character (+) always prefixes a numeric value with a sign, the space flag has no effect when combined with the plus flag.

#	For `c`, `d`, `i`, and `u` conversion types, the # flag has no effect. For s conversion types, a pointer to a Pascal string, is output as a character string. For `o` conversion types, the # flag prefixes the conversion with a 0. For `x` conversion types with this flag, the conversion is prefixed with a 0x. For `e`, `E`, `f`, `g`, and `G` conversions, the # flag forces a decimal point in the output. For `g` and `G` conversions with this flag, trailing zeroes after the decimal point are not removed.
0	This flag pads zeroes on the left of the conversion. It applies to `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversion types. The leading zeroes follow sign and base indication characters, replacing what would normally be space characters. The minus sign flag character overrides the `0` flag character. The `0` flag is ignored when used with a precision width for `d`, `i` , `o`, `u`, `x`, and `X` conversion types.

	Conversions
d	The corresponding argument is converted to a signed decimal.
i	The corresponding argument is converted to a signed decimal.
o	The argument is converted to an unsigned octal.
u	The argument is converted to an unsigned decimal.
x, X	The argument is converted to an unsigned hexadecimal. The `x` conversion type uses lowercase letters (`abcdef`) while `X` uses uppercase letters (`ABCDEF`).
n	This conversion type stores the number of items output by `printf()` so far. Its corresponding argument must be a pointer to an `int`.

f	The corresponding floating point argument (`float`, or `double`) is printed in decimal notation. The default precision is 6 (6 digits after the decimal point). If the precision width is explicitly 0, the decimal point is not printed.
e, E	The floating point argument (`float` or `double`) is output in scientific notation: [-]*b*.*aaa*e±E*ee*. There is one digit (*b*) before the decimal point. Unless indicated by an optional precision width, the default is 6 digits after the decimal point (*aaa*). If the precision width is 0, no decimal point is output. The exponent (*ee*) is at least 2 digits long. The `e` conversion type uses lowercase `e` as the exponent prefix. The `E` conversion type uses uppercase `E` as the exponent prefix.
g, G	The `g` conversion type uses the `f` or `e` conversion types and the `G` conversion type uses the `f` or `E` conversion types. Conversion type `e` (or `E`) is used only if the converted exponent is less than -4 or greater than the precision width. The precision width indicates the number of significant digits. No decimal point is output if there are no digits following it.
c	The corresponding argument is output as a character.
s	The corresponding argument, a pointer to a character array, is output as a character string. Character string output is completed when a null character is reached. The null character is not output.
p	The corresponding argument is taken to be a pointer. The argument is output using the `X` conversion type format.

	CodeWarrior Extensions
`#s`	The corresponding argument, a pointer to a Pascal string, is output as a character string. A Pascal character string is a length byte followed by the number characters specified in the length byte. **Note**: This conversion type is an extension to the ANSI C library but applied in the same manner as for other format variations.

Return `printf()`, like `fprintf()`, `sprintf()`, `vfprintf()`, and `vprintf()`, returns the number of arguments that were successfully output. `printf()` returns a negative value if it fails.

See Also **`stdio.h`**: `fprintf(), sprintf(), vprintf(), vprintf()`

Listing 18.24 Example of printf() usage.

```
#include <stdio.h>

void main(void)
{
  int i = 25;
  char c = 'M';
  short int d = 'm';
  static char s[] = "Metrowerks!";
  static char pas[] = "\pMetrowerks again!";
  float f = 49.95;
  double x = 1038.11005;
  int count;
  printf("%s printf() demonstration:\n%n", s, &count);
  printf("The last line contained %d characters\n",count);
  printf("Pascal string output: %#20s\n", pas);
  printf("%-4d %x %06x %-5o\n", i, i, i, i);
  printf("%*d\n", 5, i);
  printf("%4c %4u %4.10d\n", c, c, c);
  printf("%4c %4hu %3.10hd\n", d, d, d);
  printf("$%5.2f\n", f);
```

```
  printf("%5.2f\n%6.3f\n%7.4f\n", x, x, x);
  printf("%*.*f\n", 8, 5, x);
}
```

```
Output:
Metrowerks! printf() demonstration:
The last line contained 36 characters
Pascal string output:    Metrowerks again!
25   19 000019 31
   25
   M   77 0000000077
   m  109 0000000109
$49.95
1038.11
1038.110
1038.1101
1038.11005
```

putc

Description Write a character to a stream.

Prototype

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

Remarks The `putc()` function outputs `c` to `stream` and advances `stream`'s file position indicator.

The `putc()` works identically to the `fputc()` function, except that it is written as a macro.

Return `putc()` returns the character written when successful and return `EOF` when it fails.

See Also **`stdio.h`**: `fputc()`, `putchar()`

Listing 18.25 Example of putc() usage.

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

void main(void)
{
  FILE *f;
  static char filename[] = "checkputc";
  static char test[] = "flying fish and quail eggs";
  int i;

  // create a new file for output
  if (( f = fopen(filename, "w")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }

  // output the test character array
  // one character at a time using putc()
  for (i = 0; test[i] > 0; i++)
    putc(test[i], f);

  // close the file
  fclose(f);
}
```

putchar

Description Write a character to `stdout`.

Prototype

```
#include <stdio.h>
int putchar(int c);
```

Remarks The `putchar()` function writes character `c` to `stdout`.

Return `putchar()` returns `c` if it is successful and returns `EOF` if it fails.

See Also **`stdio.h`**: `fputc(), putc()`

Listing 18.26 Example of putchar() usage.

```
#include <stdio.h>

void main(void)
{
  static char test[] = "running jumping walking tree\n";
  int i;

  // output the test character one character
  // at a time until the null character is found.
  for (i = 0; test[i] != '\0'; i++)
    putchar(test[i]);

}
```

```
Output:
running jumping walking tree
```

puts

Description Write a character string to `stdout`.

Prototype

```
#include <stdio.h>
int puts(const char *s);
```

Remarks The `puts()` function writes a character string array to `stdout`, stopping at, but not including the terminating null character. The function also appends a newline (`'\n'`) to the output.

Return `puts()` returns zero if successful and returns a nonzero value if it fails.

See Also **stdio.h**: fputs()

Listing 18.27 Example of puts() usage.

```
#include <stdio.h>

void main(void)
{
  static char s[] = "car bus metro werks";
  int i;

  // output the string 10 times
  for (i = 0; i < 10; i++)
    puts(s);
}
```

```
Output:
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
```

remove

Description Delete a file.

Prototype

```
#include <stdio.h>
int remove(const char *filename);
```

Remarks The `remove()` function deletes the named file specified by `filename`.

Return `remove()` returns 0 if the file deletion is successful, and returns a nonzero value if it fails.

See Also **`stdio.h`**: `fopen()`, `rename()`

Listing 18.28 Example of remove() usage.

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

void main(void)
{
  char filename[40];

  // get a filename from the user
  printf("Enter the name of the file to delete.\n");
  gets(filename);

  // delete the file
  if (remove(filename) != 0) {
    printf("Can't remove %s.\n", filename);
    exit(1);
  }
}
```

rename

Description Change the name of a file.

Prototype

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Remarks The `rename()` function changes the name of a file, specified by `old` to the name specified by `new`.

Return `rename()` returns a nonzero if it fails and returns zero if successful

See Also **`stdio.h`**: `freopen(), remove()`

Listing 18.29 Example of rename() usage.

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

void main(void)
{
  char oldname[50];// current filename
  char newname[50];// new filename

  // get the current filename from the user
  printf("Please enter the current filename.\n");
  gets(oldname);

  // get the new filename from the user
  printf("Please enter the new filename.\n");
  gets(newname);

  // rename oldname to newname
  if (rename(oldname, newname) != 0) {
    printf("Can't rename %s to %s.\n", oldname,
      newname);
    exit(1);
  }
}
```

```
Output:
Please enter the current filename.
metrowerks
```

```
Please enter the new filename.
itwerks
```

rewind

Description Reset the file position indicator to the beginning of the file.

Prototype

```
#include <stdio.h>
void rewind(FILE *stream);
```

Remarks The `rewind()` function sets the file indicator position of `stream` such that the next write or read operation will be from the beginning of the file. It also undoes any previous call to `ungetc()` and clears `stream`'s end-of-file and error status.

See Also **`stdio.h`**: `fseek()`, `fsetpos()`

Listing 18.30 Example of rewind() usage.

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

void main(void)
{
  FILE *f;
  char filename[80], buf[80];

  // get a filename from the user
  printf("Enter a filename to read.\n");
  gets(filename);

  // open a file for input
  if (( f = fopen(filename, "r")) == NULL) {
    printf("Can't open %s.\n", filename);
    exit(1);
  }
```

```
  printf("Reading first line twice.\n");

  // move the file position indicator to the beginning
  // of the file
  rewind(f);
  // read the first line
  fgets(buf, 80, f);
  printf("Once: %s\n", buf);

  // move the file position indicator to the
  //beginning of the file
  rewind(f);

  // read the first line again
  fgets(buf, 80, f);
  printf("Twice: %s\n", buf);

  // close the file
  fclose(f);
}
```

```
Output:
Enter a filename to read.
itwerks
Reading first line twice.
Once: flying fish and quail eggs
Twice: flying fish and quail eggs
```

scanf

Description Read formatted text.

Prototype

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Remarks

The `scanf()` function reads text and converts the text read to programmer specified types.

The `format` argument is a character array containing normal text, white space (space, tab, newline), and conversion specifications. The normal text specifies literal characters that must be matched in the input stream. A white space character indicates that white space characters are skipped until a non-white space character is reached. The conversion specifications indicate what characters in the input stream are to be converted and stored.

The conversion specifications must have matching arguments in the order they appear in `format`. Because `scanf()` stores data in memory, the matching conversion specification arguments must be pointers to objects of the relevant types.

A conversion specification consists of the percent sign (%) prefix, followed by an optional maximum width or assignment suppression, and ending with a conversion type. A percent sign can be skipped by doubling it in format; `%%` signifies a single `%` in the input stream.

An optional width is a decimal number specifying the maximum width of an input field. `scanf()` will not read more characters for a conversion than is specified by the width.

An optional assignment suppression character (`*`) can be used to skip an item by reading it but not assigning it. A conversion specification with assignment suppression must not have a corresponding argument.

The last character, the conversion type, specifies the kind of conversion requested. Table 10 describes the conversion type characters.

Table 18.3 scanf() Format modifier types,

Modifier	Description
	Size Modifiers

h	The `h` flag indicates that the corresponding conversion modifier is a `short int` or `unsigned short int` type.
l	When used with integer conversion modifiers, the `l` flag indicates `long int` or an `unsigned long int` type. When used with floating point conversion modifier, the `l` flag indicates a `double`.
L	The `L` flag indicates that the corresponding float conversion modifier is a `long double` type.

	Conversion Modifiers
d	A decimal integer is read.
i	A decimal, octal, or hexadecimal integer is read. The integer can be prefixed with a plus or minus sign (+, -), 0 for octal numbers, 0x or 0X for hexadecimal numbers.
o	An octal integer is read.
u	An unsigned decimal integer is read.
`x`, `X`	A hexadecimal integer is read.
`e`, `E`, `f`, `g`, `G`	A floating point number is read. The number can be in plain decimal format (e.g. `3456.483`) or in scientific notation (`[-]`*b*`.`*aaa*e±*dd*).
s	A character string is read. The input character string is considered terminated when a white space character is reached or the maximum width has been reached. The null character is appended to the end of the array.
c	A character is read. White space characters are not skipped, but read using this conversion type.

p	A pointer address is read. The input format should be the same as that output by the `p` conversion type in `printf()`.
n	This conversion type does not read from the input stream but stores the number of characters read in its corresponding argument.
[scanset]	A character array is read. The *scanset* is a sequence of characters. Input stream characters are read until a character is found that is not in *scanset*. If the first character of *scanset* is a circumflex (^) then input stream characters are read until a character from *scanset* is read. A null character is appended to the end of the character array.

Return `scanf()` returns the number of items successfully read and returns `EOF` if a conversion type does not match its argument or and end-of-file is reached.

See Also **`stdio.h`**: `printf(), sscanf()`

Listing 18.31 Example of scanf() usage.

```
#include <stdio.h>

void main(void)
{
  int i;
  unsigned int j;
  char c;
  char s[40];
  double x;

  printf("Enter an integer surrounded by ! marks\n");
  scanf("!%d!", &i);
  printf("Enter three integers\n");
  printf("in hexadecimal, octal, or decimal.\n");
  // note that 3 integers are read, but only the last two
```

```
  // are assigned to i and j
  scanf("%*i %i %ui", &i, &j);

  printf("Enter a character and a character string.\n");
  scanf("%c %10s", &c, s);

  printf("Enter a floating point value.\n");
  scanf("%lf", &x);

}
```

```
Output:
Enter an integer surrounded by ! marks
!94!
Enter three integers
in hexadecimal, octal, or decimal.
1A 6 24
Enter a character and a character string.
Enter a floating point value.
A
Sounds like 'works'!
3.4
```

setbuf

Description Change the buffer size of a stream.

Prototype

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Remarks The `setbuf()` function allows the programmer to set the buffer size for `stream`. It should be called after `stream` is opened, but before it is read from or written to.

The function makes the array pointed to by `buf` the buffer used by `stream`. The `buf` argument can either be a null pointer or point to an array of size `BUFSIZ`, defined in `stdio.h`.

If `buf` is a null pointer, the stream becomes unbuffered.

See Also **stdio.h**: setvbuf(), **stdlib.h**: malloc()

Listing 18.32 Example of setbuf() usage.

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

void main(void)
{
  FILE *f;
  char name[80];

  // get a filename from the user
  printf("Enter the name of the file to write to.\n");
  gets(name);

  // create a new file for output
  if ( (f = fopen(name, "w")) == NULL) {
    printf("Can't open file %s.\n", name);
    exit(1);
  }

  setbuf(f, NULL);// turn off buffering

  // this text is sent directly to the file without
  // buffering
  fprintf(f, "Buffering is now off\n");
  fprintf(f, "for this file.\n");

  // close the file
  fclose(f);
}
```

```
Output:
Enter the name of the file to write to.
bufftest
```

setvbuf

Description Change the buffering scheme for a stream.

Prototype

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
    size_t size);
```

Remarks The `setvbuf()` allows the manipulation of the buffering scheme as well as the size of the buffer used by stream. The function should be called after the stream is opened but before it is written to or read from.

The `buf` argument is a pointer to a character array. The `size` argument indicates the size of the character array pointed to by `buf`. The most efficient buffer size is a multiple of `BUFSIZ`, defined in `stdio.h`.

If `buf` is a null pointer, then the operating system creates its own buffer of `size` bytes.

The `mode` argument specifies the buffering scheme to be used with `stream`. `mode` can have one of three values defined in `stdio.h`: `_IOFBF`, `_IOLBF`, and `_IONBF`.

`_IOFBF` specifies that `stream` be buffered.

`_IOLBF` specifies that `stream` be line buffered.

`_IONBF` specifies that `stream` be unbuffered

Return `setvbuf()` returns zero if it is successful and returns a nonzero value if it fails.

See Also **stdio.h**: setbuf(), **stdlib.h**: malloc()

Listing 18.33 Example of setvbuf() usage.

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

void main(void)
{
  FILE *f;
  char name[80];

  // get a filename from the user
  printf("Enter the name of the file to write to.\n");
  gets(name);

  // create a new file for output
  if ( (f = fopen(name, "w")) == NULL) {
    printf("Can't open file %s.\n", name);
    exit(1);
  }

  setvbuf(f, NULL, _IOLBF, 0);// line buffering
  fprintf(f, "This file is now\n");
  fprintf(f, "line buffered.\n");

  // close the file
  fclose(f);
}
```

```
Output:
Enter the name of the file to write to.
buffy
```

sprintf

Description Format a character string array.

Prototype

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Remarks The `sprintf()` function works identically to `printf()` with the addition of the `s` parameter. Output is stored in the character array pointed to by `s` instead of being sent to `stdout`. The function terminates the output character string with a null character.

For information on how to use `sprintf()` refer to the description of `printf()`.

Return `sprintf()` returns the number of characters assigned to `s`, not including the null character.

See Also **`stdio.h`**: `fprintf(), printf()`

Listing 18.34 Example of sprintf() usage.

```
#include <stdio.h>

void main(void)
{
  int i = 1;
  static char s[] = "Metrowerks";
  char dest[50];

  sprintf(dest, "%s is number %d!", s, i);
  puts(dest);
}
```

```
Output:
Metrowerks is number 1!
```

sscanf

Description Read formatted text into a character string.

Prototype

```
#include <stdio.h>
int sscanf(char *s, const char *format, ...);
```

Remarks

The `sscanf()` operates identically to `scanf()` but reads its input from the character array pointed to by `s` instead of `stdin`. The character array pointed to `s` must be null terminated.

Refer to the description of `scanf()` for more information.

Return

`scanf()` returns the number of items successfully read and converted and returns `EOF` if it reaches the end of the string or a conversion specification does not match its argument.

See Also

`stdio.h`: `fscanf(), scanf()`

Listing 18.35 Example of sscanf() usage.

```
#include <stdio.h>

void main(void)
{
  static char in[] = "figs cat pear 394 road 16!";
  char s1[20], s2[20], s3[20];
  int i;

  // get the words figs, cat, road,
  // and the integer 16
  // from in and store them in s1, s2, s3, and i,
  // respectively
  sscanf(in, "%s %s pear 394 %s %d!", s1, s2, s3, &i);
  printf("%s %s %s %d\n", s1, s2, s3, i);

}
```

```
Output:
figs cat road 16
```

tmpfile

Description Open a temporary file.

Prototype

```
#include <stdio.h>
FILE *tmpfile(void);
```

Remarks The `tmpfile()` function creates and opens a binary file that is automatically removed when it is closed or when the program terminates.

Return `tmpfile()` returns a pointer to the `FILE` variable of the temporary file if it is successful. If it fails, `tmpfile()` returns a null pointer (`NULL`).

See Also **`stdio.h`**: `fopen(), tmpnam()`

Listing 18.36 Example of tmpfile() usage.

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

void main(void)
{
  FILE *f;

  // create a new temporary file for output
  if ( (f = tmpfile()) == NULL) {
    printf("Can't open temporary file.\n");
    exit(1);
  }

  // output text to the temporary file
  fprintf(f, "watch clock timer glue\n");

  // close AND DELETE the temporary file
  // using fclose()
```

```
  fclose(f);
}
```

tmpnam

Description Create a unique temporary filename.

Prototype

```
#include <stdio.h>
char *tmpnam(char *s);
```

Remarks The `tmpnam()` functions creates a valid filename character string that will not conflict with any existing filename. A program can call the function up to `TMP_MAX` times before exhausting the unique filenames `tmpnam()` generates. The `TMP_MAX` macro is defined in `stdio.h`.

The `s` argument can either be a null pointer or pointer to a character array. The character array must be at least `L_tmpnam` characters long. The new temporary filename is placed in this array. The `L_tmpnam` macro is defined in `stdio.h`.

If `s` is `NULL`, `tmpnam()` returns with a pointer to an internal static object that can be modified by the calling program.

Unlike `tmpfile()`, a file created using a filename generated by the `tmpnam()` function is not automatically removed when it is closed.

Return `tmpnam()` returns a pointer to a character array containing a unique, non-conflicting filename. If `s` is a null pointer (`NULL`), the pointer refers to an internal static object. If `s` points to a character array, `tmpnam()` returns the same pointer.

See Also **`stdio.h`**: `fopen(), tmpfile()`

Listing 18.37 Example of tmpnam() usage.

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

void main(void)
{
  FILE *f;
  char *tempname;
  int c;

  // get a unique filename
  tempname = tmpnam("tempwerks");

  // create a new file for output
  if ( (f = fopen(tempname, "w")) == NULL) {
    printf("Can't open temporary file %s.\n", tempname);
    exit(1);
  }

  // output text to the file
  fprintf(f, "shoe shirt tie trousers\n");
  fprintf(f, "province\n");

  // close the file
  fclose(f);

  // delete the file
  remove(tempname);
}
```

ungetc

Description Place a character back into a stream.

Prototype

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

Remarks The `ungetc()` function places character `c` back into `stream`'s buffer. The next read operation will read the character placed by `ungetc()`. Only one character can be pushed back into a buffer until a read operation is performed.

The function's effect is ignored when an `fseek()`, `fsetpos()`, or `rewind()` operation is performed.

Return `ungetc()` returns `c` if it is successful and returns `EOF` if it fails.

See Also **`stdio.c`**: `fseek()`, `fsetpos()`, `rewind()`

Listing 18.38 Example of ungetc() usage.

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

void main(void)
{
  FILE *f;
  int c;

  // create a new file for output and input
  if ( (f = fopen("myfoo", "w+")) == NULL) {
    printf("Can't open myfoo.\n");
    exit(1);
  }

  // output text to the file
  fprintf(f, "The quick brown fox\n");
  fprintf(f, "jumped over the moon.\n");

  // move the file position indicator
  // to the beginning of the file
  rewind(f);

  printf("Reading each character twice.\n");

  // read a character
```

```
  while ( (c = fgetc(f)) != EOF) {
    putchar(c);
  // put the character back into the stream
    ungetc(c, f);
    c = fgetc(f);// read the same character again
    putchar(c);
  }
  fclose(f);
}
```

vfprintf

Description Write formatted output to a stream.

Prototype

```
#include <stdio.h>
int vfprintf(FILE *stream,const char
*format,va_list arg);
```

Remarks The `vfprintf()` function works identically to the `fprintf()` function. Instead of the variable list of arguments that can be passed to `fprintf()`, `vfprintf()` accepts its arguments in the array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

Return `vfprintf()` returns the number of characters written or `EOF` if it failed.

See Also **`stdio.h`**: `fprintf(), printf(), stdarg.h`

Listing 18.39 Example of vfprintf() usage.

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

int fpr(FILE *, char *, ...);
```

```
void main(void)
{
  FILE *f;
  static char name[] = "foo";
  int a = 56, result;
  double x = 483.582;

  // create a new file for output
  if (( f = fopen(name, "w")) == NULL) {
    printf("Can't open %s.\n", name);
    exit(1);
  }

  // format and output a variable number of arguments
  // to the file
  result = fpr(f, "%10s %4.4f %-10d\n", name, x, a);

  // close the file
  fclose(f);
}

// fpr() formats and outputs a variable
// number of arguments to a stream using
// the vfprintf() function
int fpr(FILE *stream, char *format, ...)
{
  va_list args;
  int retval;

  va_start(args, format);// prepare the arguments
  retval = vfprintf(stream, format, args);
  // output them
  va_end(args);// clean the stack
  return retval;
}
```

vprintf

Description Write formatted output to `stdout`.

Prototype

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Remarks The `vprintf()` function works identically to the `printf()` function. Instead of the variable list of arguments that can be passed to `printf()`, `vprintf()` accepts its arguments in the array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

Return `vprintf()` returns the number of characters written or a negative value if it failed.

See Also **`stdio.h`**: `fprintf(), printf(), stdarg.h`

Listing 18.40 Example of vprintf() usage.

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

int pr(char *, ...);

void main(void)
{
  int a = 56;
  double f = 483.582;
  static char s[] = "Metrowerks";

  // output a variable number of arguments to stdout
  pr("%15s %4.4f %-10d*\n", s, f, a);

}

// pr() formats and outputs a variable number of arguments
// to stdout using the vprintf() function
```

```
int pr(char *format, ...)
{
  va_list args;
  int retval;
  va_start(args, format); // prepare the arguments
  retval = vprintf(format, args);
  va_end(args);// clean the stack
  return retval;
}
```

```
Output:
Metrowerks 483.5820 56        *
```

vsprintf

Description Write formatted output to a string.

Prototype

```
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list
arg);
```

Remarks The `vsprintf()` function works identically to the `sprintf()` function. Instead of the variable list of arguments that can be passed to `sprintf()`, `vsprintf()` accepts its arguments in the array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

Return `vsprintf()` returns the number of characters written to `s` or `EOF` if it failed.

See Also **`stdio.h`**: `printf()`, `sprintf()`, **`stdarg.h`**

Listing 18.41 Example of vsprintf() usage.

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

```
int spr(char *, char *, ...);

void main(void)
{
  int a = 56;
  double x = 1.003;
  static char name[] = "Metrowerks";
  char s[50];

  // format and send a variable number of arguments
  // to character array s
  spr(s, "%10s\n %f\n %-10d\n", name, x, a);
  puts(s);

}

// spr() formats and sends a variable number of
// arguments to a character array using the sprintf()
// function
int spr(char *s, char *format, ...)
{
  va_list args;
  int retval;

  va_start(args, format); // prepare the arguments
  retval = vsprintf(s, format, args);
  va_end(args);// clean the stack
  return retval;
}
```

```
Output:
Metrowerks
 1.003000
 56
```

19

stdlib.h

The `stdlib.h` header file provides groups of closely related functions for string conversion, pseudo-random number generation, memory management, environment communication, searching and sorting, multibyte character conversion, and integer arithmetic.

General utilities

The `stdlib.h` header file provides groups of closely related functions for string conversion, pseudo-random number generation, memory management, environment communication, searching and sorting, multibyte character conversion, and integer arithmetic.

The string conversion functions are `atof()`, `atoi()`, `atol()`, `strtod()`, `strtol()`, and `strtoul()`.

The pseudo-random number generation functions are `rand()` and `srand()`.

The memory management functions are `calloc()`, `free()`, `malloc()`, and `realloc()`.

The environment communication functions are `abort()`, `atexit()`, `exit()`, `getenv()`, and `system()`.

The searching and sorting functions are `bsearch()` and `qsort()`.

The multibyte conversion functions convert locale-specific multibyte characters to `wchar_t` type characters (defined in `stddef.h`). The functions are `mblen()`, `mbstowcs()`, `mbtowc()`, `wcstombs()`, and `wctomb()`.

The integer arithmetic functions are `abs()`, `div()`, `labs()`, and `ldiv()`.

Many of the `stdlib.h` functions use the `size_t` type and the `NULL` macro, which are defined in `stdlib.h`.

abort

Description Abnormal program termination.

Prototype

```
#include <stdlib.h>
void abort(void)
```

Remarks The `abort()` function raises the `SIGABRT` signal and quits the program to return to the operating system.

The `abort()` function will not terminate the program if a programmer-installed signal handler uses `longjmp()` instead of returning normally.

See Also **`assert.h`**: `assert()`, **`setjmp.h`**: `longjmp()`, **`signal.h`**: `signal()`, `raise()`, **`stdlib.h`**: `atexit()`, `exit()`

Listing 19.1 Example of abort() usage.

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

void main(void)
{
  char c;

  printf("Aborting the program.\n");
  printf("Press return.\n");

  // wait for the return key to be pressed
  c = getchar();

  // abort the program
  abort();
}
```

```
Output:
Aborting the program.
Press return.
```

abs

Description Compute the absolute value of an integer.

Prototype

```
#include <stdlib.h>
int abs(int i);
```

Return `abs()` returns the absolute value of its argument. Note that the two's complement representation of the smallest negative number has no matching absolute integer representation.

See Also **math.h**: fabs(), **stdlib.h**: labs()

Listing 19.2 Example of abs() usage.

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

void main(void)
{
  int i = -20;
  long int j = -48323;

  printf("Absolute value of %d is %d.\n", i, abs(i));
  printf("Absolute value of %ld is %ld.\n", j, labs(j));

}
```

```
Output:
Absolute value of -20 is 20.
Absolute value of -48323 is 48323.
```

atexit

Description Install a function to be executed at a program's exit.

Prototype

```
#include <stdlib.h>
int atexit(void (*func) void));
```

Remarks The `atexit()` function adds the function pointed to by `func` to a list. When `exit()` is called, each function on the list is called in the reverse order in which they were installed with `atexit()`. After all the functions on the list have been called, `exit()` terminates the program.

The `stdio.h` library, for example, installs its own exit function using `atexit()`. This function flushes all buffers and closes all open streams.

Return `atexit()` returns a zero when it succeeds in installing a new exit function and returns a nonzero value when it fails.

See Also **stdlib.h**: exit()

Listing 19.3 Example of atexit() usage.

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

// Prototypes
void first(void);
void second(void);
void third(void);
```

```
void main(void)
{
  atexit(first);
  atexit(second);
  atexit(third);

  exit(0);
}


void first(void)
{
  int c;

  printf("First exit function.\n");
  printf("Press return.\n");
// wait for the return key to be pressed
  c = getchar();
}

void second(void)
{
  int c;

  printf("Second exit function.\n");
  printf("Press return.\n");
  c = getchar();
}

void third(void)
{
  int c;

  printf("Third exit function.\n");
  printf("Press return.\n");
  c = getchar();
}
```

```
Output:
Third exit function.
Press return.

Second exit function.
Press return.

First exit function.
Press return.
```

atof, atoi, atol

Description Convert a character string to a numeric value.

Prototype

```
#include <stdlib.h>
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
```

Remarks The `atof()` function converts the character array pointed to by `nptr` to a floating point value of type `double`.

The `atoi()` function converts the character array pointed to by `nptr` to an integer value.

The `atol()` function converts the character array pointed to by `nptr` to an integer of type `long int`.

All three functions skip leading white space characters.

All three functions set the global variable `errno` to `ERANGE` if the converted value cannot be expressed in their respective type.

Return `atof()` returns a floating point value of type `double`.

`atoi()` returns an integer value of type `int`.

`atol()` returns an integer value of type `long int`.

See Also errno.h, **stdio.h**: scanf()

Listing 19.4 Example of atof(), atoi(), atol() usage.

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

void main(void)
{
  int i;
  long int j;
  float f;
  static char si[] = "-493", sli[] = "63870";
  static char sf[] = "1823.4034";

  f = atof(sf);
  i = atoi(si);
  j = atol(sli);

  printf("%f %d %ld\n", f, i, j);

}
```

```
Output:
1823.403400 -493 63870
```

bsearch

Description Efficient sorted array searching.

Prototype

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
    size_t num, size_t size,
    int (*compare) (const void *, const void *))
```

Remarks

The `bsearch()` function efficiently searches a sorted array for an item using the binary search algorithm.

The `key` argument points to the item you want to search for.

The `base` argument points to the first byte of the array to be searched. This array must already be sorted in ascending order. This order is based on the comparison requirements of the function pointed to by the `compare` argument.

The `num` argument specifies the number of array elements to search.

The `size` argument specifies the size of an array element.

The compare argument is a pointer to a programmer-supplied function. This function is used to compare the key with each individual elment of the array. That compare function takes two pointers as arguments. The first argument is the key that was passed to `bsearch`() as the first argument to `bsearch()`. The second argument is a pointer to one element of the array passed as the second argument to `bsearch()`.

For explanation we will call the arguments search_key and array_element. This compare function compares the search_key to the array element. If the search_key and the array_element are equal, the function will return zero. If the search_key is less than the array_element, the function will return a negative value. If the search_key is greater than the array_element, the function will return a positive value.

Return

`bsearch()` returns a pointer to the element in the array matching the item pointed to by `key`. If no match was found, `bsearch()` returns a null pointer (`NULL`).

See Also

stdlib.h: qsort()

Listing 19.5 Example of bsearch usage.

```
// A simple telephone directory manager
// This program accepts a list of names and
```

```
// telephone numbers, sorts the list, then
// searches for specified names.

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

// Maximum number of records in the directory.
#define MAXDIR 40

typedef struct
{
  char lname[15];// keyfield--see comp() function
  char fname[15];
  char phone[15];
} DIRENTRY; // telephone directory record

int comp(const DIRENTRY *, const DIRENTRY *);
DIRENTRY *look(char *);
DIRENTRY directory[MAXDIR];// the directory itself
int reccount; // the number of records entered

void main(void)
{
  DIRENTRY *ptr;
  int lastlen;
  char lookstr[15];

  printf("Telephone directory program.\n");
  printf("Enter blank last name when done.\n");

  reccount = 0;
  ptr = directory;
  do {
    printf("\nLast name: ");
    gets(ptr->lname);
    printf("First name: ");
    gets(ptr->fname);
    printf("Phone number: ");
    gets(ptr->phone);
```

```
    if ( (lastlen = strlen(ptr->lname)) > 0) {
      reccount++;
      ptr++;
    }
  } while ( (lastlen > 0) && (reccount < MAXDIR) );

  printf("Thank you.  Now sorting. . .\n");

  // sort the array using qsort()
  qsort(directory, reccount,
        sizeof(directory[0]),comp);

  printf("Enter last name to search for,\n");
  printf("blank to quit.\n");
  printf("\nLast name: ");
  gets(lookstr);

  while ( (lastlen = strlen(lookstr)) > 0) {
    ptr = look(lookstr);
    if (ptr != NULL)
      printf("%s, %s: %s\n",
        ptr->lname,
        ptr->fname,
        ptr->phone);
    elseprintf("Can't find %s.\n", lookstr);
    printf("\nLast name: ");
    gets(lookstr);
  }

  printf("Done.\n");

}

int comp(const DIRENTRY *rec1, const DIRENTRY *rec2)
{
  return (strcmp((char *)rec1->lname,
             (char *)rec2->lname));
}

// search through the array using bsearch()
```

```
DIRENTRY *look(char k[])
{
  return (DIRENTRY *) bsearch(k, directory, reccount,
sizeof(directory[0]), comp);
}
```

Output

```
Telephone directory program.
Enter blank last name when done.

Last name: Mation
First name: Infor
Phone number: 555-1212

Last name: Bell
First name: Alexander
Phone number: 555-1111

Last name: Johnson
First name: Betty
Phone number: 555-1010

Last name:
First name:
Phone number:
Thank you.  Now sorting. . .
Enter last name to search for,
blank to quit.

Last name: Mation
Infor, Mation: 555-1212

Last name: Johnson
Johnson, Betty: 555-1010

Last name:
Done.
```

calloc

Description Allocate space for a group of objects.

Prototype

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Remarks The `calloc()` function allocates contiguous space for `nmemb` elements of size `size`. The space is initialized with zeroes.

Return `calloc()` returns a pointer to the first byte of the memory area allocated. `calloc()` returns a null pointer (`NULL`) if no space could be allocated.

See Also **`stdlib.h`**: `free()`, `malloc()`, `realloc()`

Listing 19.6 Example of calloc() usage.

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

void main(void)
{
  static char s[] = "Metrowerks compilers";
  char *sptr1, *sptr2, *sptr3;

  // allocate the memory three different ways
  // one: allocate a thirty byte block of
  //uninitialized memory
  sptr1 = (char *) malloc(30);
  strcpy(sptr1, s);
  printf("Address of sptr1: %p\n", sptr1);

  // two: allocate twenty bytes of unitialized memory
  sptr2 = (char *) malloc(20);
  printf("sptr2 before reallocation: %p\n", sptr2);
  strcpy(sptr2, s);
```

```
  // now re-allocate ten extra bytes (for a total of
  // thirty bytes)
  //
  // note that the memory block pointed to by sptr2 is
  // still contiguous after the call to realloc()
  sptr2 = (char *) realloc(sptr2, 30);
  printf("sptr2 after reallocation: %p\n", sptr2);

  // three: allocate thirty bytes of initialized memory
  sptr3 = (char *) calloc(strlen(s), sizeof(char));
  strcpy(sptr3, s);
  printf("Address of sptr3: %p\n", sptr3);

  puts(sptr1);
  puts(sptr2);
  puts(sptr3);

  // release the allocated memory to the heap
  free(sptr1);
  free(sptr2);
  free(sptr3);

}
```

```
Output:
Address of sptr1: 5e5432
sptr2 before reallocation: 5e5452
sptr2 after reallocation: 5e5468
Address of sptr3: 5e5488
Metrowerks compilers
Metrowerks compilers
Metrowerks compilers
```

div

Description Compute the integer quotient and remainder.

Prototype

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Remarks The `div_t` type is defined in `stdlib.h` as

```
typedef struct { int quot,rem; } div_t;
```

Return `div()` divides `denom` into `numer` and returns the quotient and remainder as a `div_t` type.

See Also **math.h**: fmod(), **stdlib.h**: ldiv()

Listing 19.7 Example of div() usage.

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

void main(void)
{
  div_t result;
  ldiv_t lresult;

  int d = 10, n = 103;
  long int ld = 1000L, ln = 1000005L;

  result = div(n, d);
  lresult = ldiv(ln, ld);

  printf("%d / %d has a quotient of %d\n",
         n, d, result.quot);
  printf("and a remainder of %d\n", result.rem);
  printf("%ld / %ld has a quotient of %ld\n",
         ln, ld, lresult.quot);
  printf("and a remainder of %ld\n", lresult.rem);
}
```

```
Output:
103 / 10 has a quotient of 10
and a remainder of 3
```

```
1000005 / 1000 has a quotient of 1000
and a remainder of 5
```

exit

Description Terminate a program normally.

Prototype

```
#include <stdlib.h>
void exit(int status);
```

Remarks The `exit()` function calls every function installed with `atexit()` in the reverse order of their installation, flushes the buffers and closes all open streams, then calls the Toolbox system call `ExitToShell`.

Return `exit()` does not return any value to the operating system. The `status` argument is kept to conform to the ANSI C Standard Library specification.

See Also **stdlib.h**: abort(), atexit()

Listing 19.8 Example of exit() usage.

```
#include <stdlib.h>

void main(void)
{
  // exit from the program
  // note: the argument is ignored by Metrowerks C
  exit(0);
}
```

free

Description Release previously allocated memory to heap.

Prototype

```
#include <stdlib.h>
void free(void *ptr);
```

Remarks The `free()` function releases a previously allocated memory block, pointed to by `ptr`, to the heap. The `ptr` argument should hold an address returned by the memory allocation functions `calloc()`, `malloc()`, or `realloc()`. Once the memory block `ptr` points to has been released, it is no longer valid. The `ptr` variable should not be used to reference memory again until it is assigned a value from the memory allocation functions.

See Also **stdlib.h**: calloc(), malloc(), realloc()

Listing 19.9 For example of free() usage

Refer to“Example of calloc() usage.” on page 184 .

getenv

Description Environment list access.

Prototype

```
#include <stdlib.h>
char *getenv(const char *name);
```

Remarks The `getenv()` is an empty function that always returns a null pointer (`NULL`). It is included in the Metrowerks `stdlib.h` header file to conform to the ANSI C Standard Library specification.

Return `getenv()` always returns a null pointer (`NULL`).

See Also **stdlib.h**: system()

labs

Description Compute long integer absolute value.

Prototype
```
#include <stdlib.h>
long int labs(long int j);
```

Return `labs()` returns the absolute value of its argument as a `long int` type.

See Also **math.h**: fabs(), **stdlib.h**: abs()

Listing 19.10 For example of labs() usage

Refer to “Example of abs() usage.” on page 175.

ldiv

Description Compute the long integer quotient and remainder.

Prototype
```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Remarks The `ldiv_t` type is defined in `stdlib.h` as
```
typedef struct {
  long int quot, rem;
} ldiv_t;
```

Return `ldiv()` divides `denom` into `numer` and returns the quotient and remainder as an `ldiv_t` type.

See Also **math.h**: fmod(), **stdlib.h**: div()

Listing 19.11 For example of ldiv() usage

```
Refer to “Example of div() usage.” on page 186 .
```

malloc

Description Allocate a block of heap memory.

Prototype

```
#include <stdlib.h>
void *malloc(size_t size);
```

Remarks The `malloc()` function allocates a block of contiguous heap memory `size` bytes large.

Return `malloc()` returns a pointer to the first byte of the allocated block if it is successful and return a null pointer if it fails.

See Also **stdlib.h**: calloc(), free(), realloc()

Listing 19.12 For example of malloc() usage

```
Refer to “Example of calloc() usage.” on page 184.
```

mblen

Description Compute the length of a multibyte character.

Prototype

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Remarks The `mblen()` function returns the length of the multibyte character pointed to by `s`. It examines a maximum of `n` characters.

The Metrowerks C implementation supports the `"C"` locale only and returns the value of `mbtowc(NULL, s, n)`.

Return `mblen()` returns the value of `mbtowc(NULL, s, n)`.

See Also locale.h, **stdlib.h**: mbtowc()

mbstowcs

Description Convert a multibyte character array to a `wchar_t` array.

Prototype

```
#include <stddlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s,
size_t n);
```

Remarks The `mbstowcs()` function converts a character array containing multibyte characters to a character array containing `wchar_t` type characters. The `wchar_t` type is defined in `stddef.h`.

The Metrowerks C implementation of `mbstowcs()` performs no translation; it copies a maximum of `n` bytes from the array pointed to by `s` to the array pointed to by `pwcs`. The function terminates prematurely if a null character is reached.

Return `mbstowcs()` returns the number of bytes copied from `s` to `pwcs`.

See Also locale.h, **stdlib.h**: wcstombs()

mbtowc

Description Translate a multibyte character to a `wchar_t` type.

Prototype

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Remarks The `mbtowc()` function converts a multibyte character, pointed to by `s`, to a character of type `wchar_t`, pointed to by `pwc`. The function converts a maximum of `n` bytes.

The Metrowerks C implementation performs no translation; it copies the first character at `s` to the first character at `pwc`.

Return `mbtowc()` returns -1 if `n` is zero and `s` is not a null pointer.

`mbtowc()` returns 0 if `s` is a null pointer or s points to a null character (`'\0'`).

`mbtowc()` returns 1 if `s` is not a null pointer and it does not point to a null character (`'\0'`).

See Also locale.h, **stdlib.h:** mblen(), wctomb()

qsort

Description Sort an array.

Prototype

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int (*compare) (const void *, const void *))
```

Remarks The `qsort()` function sorts an array using the quicksort algorithm. It sorts the array without displacing it; the array occupies the same memory it had before the call to `qsort()`.

The `base` argument is a pointer to the base of the array to be sorted.

The `nmemb` argument specifies the number of array elements to sort.

The `size` argument specifies the size of an array element.

The `compare` argument is a pointer to a programmer-supplied compare function. The function takes two pointers to different array elements and compares them based on the key. If the two elements are equal, `compare` must return a zero. The `compare` function must re-

turn a negative number if the first element is less than the second. Likewise, the function must return a positive number if the first argument is greater than the second.

See Also **stdlib.h**: bsearch()

Listing 19.13 **For example of qsort() usage**

Refer to “Example of bsearch usage.” on page 180 .

rand

Description Generate a pseudo-random integer value.

Prototype

```
#include <stdlib.h>
int rand(void);
```

Remarks A sequence of calls to the `rand()` function generates and returns a sequence of pseudo-random integer values from 0 to `RAND_MAX`. The `RAND_MAX` macro is defined in `stdlib.h`.

By seeding the random number generator using `srand()`, different random number sequences can be generated with `rand()`.

Return `rand()` returns a pseudo-random integer value between 0 and `RAND_MAX`.

See Also **stdlib.h**: srand()

Listing 19.14 **Example of rand() usage.**

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

```
void main(void)
{
  int i;
  unsigned int seed;

  for (seed = 1; seed <= 5; seed++) {
    srand(seed);
    printf("First five random number for seed %d:\n",
        seed);
    for (i = 0; i < 5; i++)
      printf("%10d", rand());
    printf("\n\n");// terminate the line
  }

}
```

```
Output:
First five random number for seed 1:
     16838      5758     10113     17515     31051

First five random number for seed 2:
       908     22817     10239     12914     25837

First five random number for seed 3:
     17747      7107     10365      8312     20622

First five random number for seed 4:
      1817     24166     10491      3711     15407

First five random number for seed 5:
     18655      8457     10616     31877     10193
```

realloc

Description Change the size of an allocated block of heap memory.

Prototype

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

```
void *realloc(void *ptr, size_t size);
```

Remarks The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The `size` argument can have a value smaller or larger than the current size of the block `ptr` points to. The `ptr` argument should be a value assigned by the memory allocation functions `calloc()` and `malloc()`.

If `size` is 0, the memory block pointed to by `ptr` is released. If `ptr` is a null pointer, `realloc()` allocates `size` bytes.

The old contents of the memory block are preserved in the new block if the new block is larger than the old. If the new block is smaller, the extra bytes are cut from the end of the old block.

Return `realloc()` returns a pointer to the new block if it is successful and `size` is greater than 0. `realloc()` returns a null pointer if it fails or `size` is 0.

See Also **stdlib.h**: calloc(), free(), malloc()

Listing 19.15 For example of realloc() usage

Refer to “Example of calloc() usage.” on page 184.

srand

Description Set the pseudo-random number generator seed.

Prototype

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Remarks The `srand()` function sets the seed for the pseudo-random number generator to `seed`. Each seed value produces the same sequence of random numbers when it is used.

See Also **stdlib.h**: rand()

Listing 19.16 **For example of labs() usage**

Refer to “Example of rand() usage.” on page 193.

strtod, strtol, strtoul

Description Character array to numeric conversions.

Prototype

```
#include <stdlib.h>
double strtod
    (const char *nptr, char **endptr);
long int strtol
    (const char *nptr, char **endptr, int base);
unsigned long int strtoul
    (const char *nptr, char **endptr, int base);
```

Remarks The `strtod()` converts a character array, pointed to by `nptr`, to a floating point value of type `double`. The character array can be in either decimal notation (e.g. `103.578`) or scientific notation (`[-]`*b*`.`*aaa*`e±E`*ee*).

The `strtol()` function converts a character array, pointed to by `nptr`, to an integer value of type `long int`, in base `base`. A plus or minus sign (+ or -) prefixing the number string is optional.

The `strtoul()` function converts a character array, pointed to by `nptr`, to an integer value of type `unsigned long int`, in base `base`. A plus or minus sign prefix is ignored.

The `base` argument in `strtol()` and `strtoul()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. If `base` is 0, then `strtol()` and `strtoul()` convert the character array based on its format. Character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'`

are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to the functions' respective types.

All three functions skip leading white space.

All three functions set the global variable `errno` to `ERANGE` if there is a conversion error.

Return

`strtod()` returns a floating point value of type `double`. If `nptr` cannot be converted to an expressible double value, `strtod()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`.

`strtol()` returns an integer value of type `long int`. If the converted value is less than `LONG_MIN`, `strtol()` returns `LONG_MIN` and sets `errno` to `ERANGE`. If the converted value is greater than `LONG_MAX`, `strtol()` returns `LONG_MAX` and sets `errno` to `ERANGE`. The `LONG_MIN` and `LONG_MAX` macros are defined in `limits.h`.

`strtoul()` returns an unsigned integer value of type `unsigned long int`. If the converted value is greater than `ULONG_MAX`, `strtoul()` returns `ULONG_MAX` and sets `errno` to `ERANGE`. The `ULONG_MAX` macro is defined in `limits.h`

See Also

errno.h, limits.h, math.h, **stdio.h**: scanf()

Listing 19.17 Example of strtod(), strtol(), strtoul() usage.

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

void main(void)
{
  double f;
  long int i;
```

```
  unsigned long int j;
  static char si[] = "4733!", sf[] = "103.749?";
  static char sb[] = "0x10*";
  char *endptr;

  f = strtod(sf, &endptr);
  printf("%f %c\n", f, *endptr);

  i = strtol(si, &endptr, 10);
  printf("%ld %c\n", i, *endptr);

  i = strtol(si, &endptr, 8);
  printf("%ld %c\n", i, *endptr);

  j = strtoul(sb, &endptr, 0);
  printf("%ld %c\n", j, *endptr);


  j = strtoul(sb, &endptr, 10);
  printf("%ld %c\n", j, *endptr);

}
```

```
Output:
103.749000 ?
4733 !
2523 !
16 *
0 x
```

system

Description Environment list assignment.

Prototype

```
#include <stdlib.h>
int system(const char *string);
```

Remarks The `system()` function is an empty function that is included in the Metrowerks `stdlib.h` to conform to the ANSI C Standard Library specification.

Return `system()` always returns 0.

See Also **stdlib.h**: getenv()

wcstombs

Description Translate a `wchar_t` type character array to a multibyte character array.

Prototype

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs,
size_t n);
```

Remarks The `wcstombs()` function converts a character array containing `wchar_t` type characters to a character array containing multibyte characters. The `wchar_t` type is defined in `stddef.h`.

The Metrowerks C implementation of `wcstombs()` performs no translation; it copies a maximum of `n` bytes from the array pointed to by `pwcs` to the array pointed to by `s` . The function terminates prematurely if a null character is reached.

Return `wcstombs()` returns the number of bytes copied from `pwcs` to `s`.

See Also **locale.h, stdlib.h**: mbstowcs()

wctomb

Description Translate a `wchar_t` type to a multibyte character.

Prototype

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Remarks

The `wctomb()` function converts a `wchar_t` type character to a multibyte character.

The Metrowerks C implementation of `wctomb()` performs no translation; it assigns `wchar` to the character pointed to by `s`.

Return

`wctomb()` returns 1 if s is not null and returns 0 otherwise.

See Also

locale.h, mbtowc()

20

string.h

The `string.h` header file provides functions for comparing, copying, concatenating, and searching character arrays and arrays of larger items.

String and array manipulation

The `string.h` header file provides functions for comparing, copying, concatenating, and searching character arrays and arrays of larger items.

The function naming convention used in `string.h` determines the type of data structure(s) a function manipulates.

A function with an `str` prefix operates on character arrays terminated with a null character (`'\0'`). The `str` functions are `strcat()`, `strcmp()`, `strchr()`, `strrchr()`, `strspn()`, `strcspn()`, `strpbrk()`, `strstr()`, `strlen()`, `strerror()`, `strtok()`, `strcoll()`, and `strxfrm()`.

A function with an `strn` prefix operates on character arrays of a length specified as a function argument. The `strn` functions are `strncpy()`, `strncat()`, and `strncmp()`.

A function with a `mem` prefix operates on arrays of items or contiguous blocks of memory. The size of the array or block of memory is specified as a function argument. The `mem` functions are `memchr()`, `memcmp()`, `memcpy()`, `memmove()`, and `memset()`.

memchr

Description Search for an occurrence of a character.

Prototype

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Remarks The `memchr()` function looks for the first occurrence of `c` in the first `n` characters of the memory area pointed to by `s`.

Return `memchr()` returns a pointer to the found character, or a null pointer (`NULL`) if `c` cannot be found.

See Also **`string.h`**: `strchr(), strrchr()`

Listing 20.1 Example of memchr() usage.

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

#define ARRAYSIZE 100

void main(void)
{
  // s1 must by same length as s2 for this example!
  static char s1[ARRAYSIZE] = "laugh* giggle 231!";
  static char s2[ARRAYSIZE] = "grunt sigh# snort!";
  char dest[ARRAYSIZE];
  char *strptr;
  int len1, len2, lendest;

  // Clear destination string using memset()
  memset( (char *)dest, '\0', ARRAYSIZE);

  // String lengths are needed by the mem functions
  // Add 1 to include the terminating '\0' character
  len1 = strlen(s1) + 1;
  len2 = strlen(s2) + 1;
  lendest = strlen(dest) + 1;

  printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

  if (memcmp( (char *)s1, (char *)s2, len1) > 0)
```

```
    memcpy( (char *)dest, (char *)s1, len1);
  else
    memcpy( (char *)dest, (char *)s2, len2);

  printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

  // copy s1 onto itself using memchr() and memmove()
  strptr = (char *)memchr( (char *)s1, '*', len1);
  memmove( (char *)strptr, (char *)s1, len1);

  printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

}
```

```
Output:
s1=laugh* giggle 231!
 s2=grunt sigh# snort!
 dest=

 s1=laugh* giggle 231!
 s2=grunt sigh# snort!
 dest=laugh* giggle 231!

 s1=laughlaugh* giggle 231!
 s2=grunt sigh# snort!
 dest=laugh* giggle 231!
```

memcmp

Description Compare two blocks of memory.

Prototype

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t
n);
```

Remarks The `memcmp()` function compares the first `n` characters of `s1` to `s2` one character at a time.

Return `memcmp()` returns a zero if all `n` characters pointed to by `s1` and `s2` are equal.

`memcmp()` returns a negative value if the first non-matching character pointed to by `s1` is less than the character pointed to by `s2`.

`memcmp()` returns a positive value if the first non-matching character pointed to by `s1` is greater than the character pointed to by `s2`.

See Also **`string.h`**: `strcmp(), strncmp()`

Listing 20.2 For example of memcmp() usage

Refer to“Example of memchr() usage.” on page 202.

memcpy

Description Copy a contiguous memory block.

Prototype

```
#include <string.h>
void *memcpy(const void *dest, const void *source,
    size_t n);
```

Remarks The `memcpy()` function copies the first `n` characters from the item pointed to by `source` to the item pointed to by `dest`. The behavior of `memcpy()` is undefined if the areas pointed to by `dest` and `source` overlap. The `memmove()` function reliably copies overlapping memory blocks.

Return `memcpy()` returns the value of `dest`.

See Also **`string.h`**: `memmove(), strcpy(), strncpy()`

Listing 20.3 For example of memcpy() usage

```
Refer to  “Example of memchr() usage.” on page 202.
```

memmove

Description Copy an overlapping contiguous memory block.

Prototype

```
#include <string.h>
void *memmove(void *dest, const void *source,
size_t n);
```

Remarks The `memmove()` function copies the first `n` characters of the item pointed to by `source` to the item pointed to by `dest`.

Unlike `memcpy()`, the `memmove()` function safely copies overlapping memory blocks.

Return `memmove()` returns the value of `dest`.

See Also **`string.h`**: `memcpy(), memset(), strcpy(), strncpy()`

Listing 20.4 For example of memmove() usage

```
Refer to “Example of memchr() usage.” on page 202.
```

memset

Description Clear the contents of a block of memory.

Prototype

```
#include <string.h>
void *memset(void *dest, int c, size_t n);
```

Remarks The `memset()` function assigns `c` to the first `n` characters of the item pointed to by `dest`.

Return `memset()` returns the value of `dest`.

Listing 20.5 For example of memset() usage

```
Refer to  “Example of memchr() usage.” on page 202 .
```

strcat

Description Concatenate two character arrays.

Prototype

```
#include <string.h>
char *strcat(char *dest, const char *source);
```

Remarks The `strcat()` function appends a copy of the character array pointed to by `source` to the end of the character array pointed to by `dest`. The `dest` and `source` arguments must both point to null terminated character arrays. `strcat()` null terminates the resulting character array.

Return `strcat()` returns the value of dest.

See Also **`string.h`**: `strncat()`

Listing 20.6 Example of strcat() usage.

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

void main(void)
{
  char s1[100] = "The quick brown fox ";
  static char s2[] = "jumped over the lazy dog.";
```

```
  strcat(s1, s2);
  puts(s1);

}
```

```
Output:
The quick brown fox jumped over the lazy dog.
```

strchr

Description Search for an occurrence of a character.

Prototype

```
#include <string.h>
char *strchr(const char *s, int c);
```

Remarks The `strchr()` function searches for the first occurrence of the character `c` in the character array pointed to by `s`. The `s` argument must point to a null terminated character array.

Return `strchr()` returns a pointer to the successfully located character. If it fails, `strchr()` returns a null pointer (`NULL`).

See Also **`string.h`**: `memchr()`, `strrchr()`

Listing 20.7 Example of strchr() usage.

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

void main(void)
{
  static char s[] = "tree * tomato eggplant garlic";
  char *strptr;

  strptr = strchr(s, '*');
```

```
  puts(strptr);

}
```

```
Output:
* tomato eggplant garlic
```

strcmp

Description Compare two character arrays.

Prototype

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Remarks The `strcmp()` function compares the character array pointed to by `s1` to the character array pointed to by `s2`. Both `s1` and `s2` must point to null terminated character arrays.

Return `strcmp()` returns a zero if `s1` and `s2` are equal, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

See Also **`string.h`**: `memcmp(), strcoll(), strncmp()`

Listing 20.8 Example of strcmp() usage.

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

void main (void)
{
  static char s1[] = "butter", s2[] = "olive oil";
  char dest[20];

  if (strcmp(s1, s2) < 0)
    strcpy(dest, s2);
  else
```

```
    strcpy(dest, s1);

  printf(" s1=%s\n s2=%s\n dest=%s\n", s1, s2, dest);

}
```

```
Output:
 s1=butter
 s2=olive oil
 dest=olive oil
```

strcpy

Description Copy one character array to another.

Prototype

```
#include <string.h>
char *strcpy(char  *dest, const char *source);
```

Remarks The `strcpy()` function copies the character array pointed to by `source` to the character array pointed to `dest`. The `source` argument must point to a null terminated character array. The resulting character array at `dest` is null terminated as well.

If the arrays pointed to by `dest` and `source` overlap, the operation of `strcpy()` is undefined.

Return `strcpy()` returns the value of dest.

See Also **`string.h`**: `memcpy()`, `memmove()`, `strncpy()`

Listing 20.9 Example of strcpy() usage.

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

void main(void)
```

```
{
  char d[30] = "";
  static char s[] = "Metrowerks";

  printf(" s=%s\n d=%s\n", s, d);
  strcpy(d, s);
  printf(" s=%s\n d=%s\n", s, d);

}
```

```
Output:
 s=Metrowerks
 d=
 s=Metrowerks
 d=Metrowerks
```

strcoll

Description Compare two character arrays according to locale.

Prototype

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Remarks The `strcoll()` function compares two character arrays based on the collating sequence set by the `locale.h` header file.

The Metrowerks C implementation of `strcoll()` compares two character arrays using `strcmp()`. It is included in the string library to conform to the ANSI C Standard Library specification.

Return `strcoll()` returns zero if `s1` is equal to `s2`, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

See Also `locale.h`, **`string.h`**: `memcmp()`, `strcmp()`, `strncmp()`

Listing 20.10 Example of strcoll() usage.

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

void main(void)
{
  static char s1[] = "aardvark", s2[] = "xylophone";
  int result;

  result = strcoll(s1, s2);

  if (result < 1)
    printf("%s is less than %s\n", s1, s2);
  else
    printf("%s is equal or greater than %s\n", s1, s2);
}
```

```
Output:
aardvark is less than xylophone
```

strcspn

Description Count characters in one character array that are not in another.

Prototype

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Remarks The `strcspn()` function counts the initial length of the character array pointed to by `s1` that does not contain characters in the character array pointed to by `s2`. The function starts counting characters at the beginning of `s1` and continues counting until a character in `s2` matches a character in `s1`.

Both `s1` and `s2` must point to null terminated character arrays.

Return `strcspn()` returns the length of characters in `s1` that does not match any characters in `s2`.

See Also **`string.h`**: `strpbrk(), strspn()`

Listing 20.11 Example of strcspn() usage.

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

void main(void)
{
  static char s1[] = "chocolate *cinnamon* 2 ginger";
  static char s2[] = "1234*";

  printf(" s1 = %s\n s2 = %s\n", s1, s2);
  printf(" %d\n", strcspn(s1, s2));
}
```

```
Output:
 s1 = chocolate *cinnamon* 2 ginger
 s2 = 1234*
 10
```

strerror

Description Return an error message in a character array.

Prototype

```
#include <string.h>
char *strerror(int errnum);
```

Remarks The `strerror()` function returns a pointer to a null terminated character array that contains an error message. The `errnum` argument has no effect on the message returned by `strerror()`; it is included to conform to the ANSI C Standard Library specification.

Return `strerror()` returns a pointer to a null terminated character array containing an error message.

Listing 20.12 Example of strerror() usage.

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

void main(void)
{
  puts(strerror(8));
}
```

```
Output:
error #008
```

strlen

Description Compute the length of a character array.

Prototype

```
#include <string.h>
size_t strlen(const char *s);
```

Remark The `strlen()` function computes the number of characters in a null terminated character array pointed to by `s`. The null character (`'\0'`) is not added to the character count.

Return `strlen()` returns the number of characters in a character array not including the terminating null character.

Listing 20.13 Example of strlen() usage.

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

```
void main(void)
{
  static char s[] = "antidisestablishmentarianism";

  printf("The length of %s is %ld.\n", s, strlen(s));

}
```

```
Output:
The length of antidisestablishmentarianism is 28.
```

strncat

Description Append a specified number of characters to a character array.

Prototype

```
#include <string.h>
char *strncat(char *dest, const char *source,
size_t n);
```

Remarks The `strncat()` function appends a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest`. The `dest` argument must point to a null terminated character array. The `source` argument does not necessarily have to point to a null terminated character array.

If a null character is reached in `source` before `n` characters have been appended, `strncat()` stops.

When done, `strncat()` terminates `dest` with a null character (`'\0'`).

Return `strncat()` returns the value of `dest`.

See Also **`string.h`**: `strcat()`

Listing 20.14 Example of strncat() usage.

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

void main(void)
{
  static char s1[100] = "abcdefghijklmnopqrstuv";
  static char s2[] = "wxyz0123456789";

  strncat(s1, s2, 4);
  puts(s1);

}
```

```
Output:
abcdefghijklmnopqrstuvwxyz
```

strncmp

Description Compare a specified number of characters.

Prototype

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t
n);
```

Remarks The `strncmp()` function compares `n` characters of the character array pointed to by `s1` to `n` characters of the character array pointed to by `s2`. Both `s1` and `s2` do not necessarily have to be null terminated character arrays.

The function stops prematurely if it reaches a null character before `n` characters have been compared.

Return `strncmp()` returns a zero if the first `n` characters of `s1` and `s2` are equal, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

See Also **string.h**: memcmp(), strcmp()

Listing 20.15 Example of strncmp() usage.

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

void main(void)
{
  static char s1[] = "12345anchor", s2[] = "12345zebra";

  if (strncmp(s1, s2, 5) == 0)
    printf("%s is equal to %s\n", s1, s2);
  else
    printf("%s is not equal to %s\n", s1, s2);

}
```

```
Output:
12345anchor is equal to 12345zebra
```

strncpy

Description Copy a specified number of characters.

Prototype

```
#include <string.h>
char *strncpy(char *dest, const char *source,
size_t n);
```

Remarks The `strncpy()` function copies a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest`. Neither `dest` nor `source` must necessarily point to null terminated character arrays. Also, `dest` and `source` must not overlap.

If a null character (`'\0'`) is reached in `source` before `n` characters have been copied, `strncpy()` continues padding `dest` with null characters until `n` characters have been added to `dest`.

The function does not terminate `dest` with a null character if `n` characters are copied from `source` before reaching a null character.

Return `strncpy()` returns the value of `dest`.

See Also **`string.h`**: `memcpy(), memmove(), strcpy()`

Listing 20.16 Example of strncpy usage.

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

void main(void)
{
  char d[50];
  static char s[] = "123456789ABCDEFG";

  strncpy(d, s, 9);
  puts(d);
}
```

```
Output:
123456789
```

strpbrk

Description Look for the first occurrence of an array of characters in another.

Prototype

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Remarks The `strpbrk()` function searches the character array pointed to by `s1` for the first occurrence of a character in the character array pointed to by `s2`.

Both `s1` and `s2` must point to null terminated character arrays.

Return `strpbrk()` returns a pointer to the first character in `s1` that matches any character in `s2`, and returns a null pointer (`NULL`) if no match was found.

See Also **`string.h`**: `strcspn()`

Listing 20.17 Example of strpbrk usage.

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

void main(void)
{
  static char s1[] = "orange banana pineapple *plum";

  static char s2[] = "*%#$";
  puts(strpbrk(s1, s2));
}
```

```
Output:
*plum
```

strrchr

Description Search for the last occurrence of a character.

Prototype

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Remarks The `strrchr()` function searches for the last occurrence of `c` in the character array pointed to by `s`. The `s` argument must point to a null terminated character array.

Return `strrchr()` returns a pointer to the character found or returns a null pointer (`NULL`) if it fails.

See Also **`string.h`**: `memchr(), strchr()`

Listing 20.18 Example of strrchr() usage.

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

void main(void)
{
  static char s[] = "Marvin Melany Metrowerks";
  puts(strrchr(s, 'M'));
}
```

```
Output:
Metrowerks
```

strspn

Description Count characters in one character array that are in another.

Prototype

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Remarks The `strspn()` function counts the initial number of characters in the character array pointed to by `s1` that contains characters in the character array pointed to by `s2`. The function starts counting characters at the beginning of `s1` and continues counting until it finds a character that is not in `s2`.

Both `s1` and `s2` must point to null terminated character arrays.

Return `strspn()` returns the number of characters in `s1` that matches the characters in `s2`.

See Also **`string.h`**: `strpbrk(), strcspn()`

Listing 20.19 Example of strspn() usage.

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

void main(void)
{
  static char s1[] = "create *build* construct";
  static char s2[] = "create *";

  printf(" s1 = %s\n s2 = %s\n", s1, s2);
  printf(" %d\n", strspn(s1, s2));
}
```

```
Output:
s1 = create *build* construct
 s2 = create *
 8
```

strstr

Description Search for a character array within another.

Prototype

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Remarks The `strstr()` function searches the character array pointed to by `s1` for the first occurrence of the character array pointed to by `s2`.

Both `s1` and `s2` must point to null terminated (`'\0'`) character arrays.

Return `strstr()` returns a pointer to the first occurrence of `s2` in `s1` and returns a null pointer (`NULL`) if `s2` cannot be found.

See Also `string.h: memchr(), strchr()`

Listing 20.20 Example of strstr() usage.

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

void main(void)
{
  static char s1[] = "tomato carrot onion";
  static char s2[] = "on";
  puts(strstr(s1, s2));
}
```

```
Output:
onion
```

strtok

Description Extract tokens within a character array.

Prototype

```
#include <string.h>
char *strtok(char *str, const char *sep);
```

Remarks The `strtok()` function tokenizes the character array pointed to by `str`. The `sep` argument points to a character array containing token separator characters. The tokens in `str` are extracted by successive calls to `strtok()`.

The first call to `strtok()` causes it to search for the first character in `str` that does not occur in `sep`. The function returns a pointer to the beginning of this first token. If no such character can be found, `strtok()` returns a null pointer (`NULL`).

If, on the first call, `strtok()` finds a token, it searches for the next token.

The function searches by skipping characters in the token in `str` until a character in `sep` is found. This character is overwritten with a null character to terminate the token string, thereby modifying the character array contents. The function also keeps its own pointer to the character after the null character for the next token. Subsequent token searches continue in the same manner from the internal pointer.

Subsequent calls to `strtok()` with a NULL `str` argument cause it to return pointers to subsequent tokens in the original `str` character array. If no tokens exist, `strtok()` returns a null pointer. The `sep` argument can be different for each call to `strtok()`.

Both `str` and `sep` must be null terminated character arrays.

Return

When first called `strtok()` returns a pointer to the first token in `str` or returns a null pointer if no token can be found.

Subsequent calls to `strtok()` with a NULL `str` argument causes `strtok()` to return a pointer to the next token or return a null pointer (`NULL`) when no more tokens exist.

`strtok()` modifies the character array pointed to by `str`.

Listing 20.21 Example of strtok() usage.

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

void main(void)
{
  static char s[50] = "(ape+bear)*(cat+dog)";
```

```
  char *nexttok;

  // first call to strtok()
  puts(strtok(s, "()+*"));

  nexttok = strtok(NULL, "()+*");
  puts(nexttok);

  nexttok = strtok(NULL, "()+*");
  puts(nexttok);

  nexttok = strtok(NULL, "()+*");
  puts(nexttok);
}
```

```
Output:
ape
bear
cat
dog
```

strxfrm

Description Transform a locale-specific character array.

Prototype

```
#include <string.h>
size_t strxfrm(char *dest,
    const char *source, size_t n);
```

Remarks The `strxfrm()` function copies characters from the character array pointed to by `source` to the character array pointed to by `dest`, transforming each character to conform to the locale character set defined in `locale.h`.

The Metrowerks C implementation of `strxfrm()` copies a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest` using the `strncpy()`

function. It is included in the string library to conform to the ANSI C Standard Library specification.

Return `strxfrm()` returns the length of `dest` after it has received `source`.

See Also `locale.h`, **`string.h`**: `strncpy()`

Listing 20.22 Example of strxfrm() usage.

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

void main(void)
{
  char d[50];
  static char s[] = "123456789ABCDEFG";
  size_t result;

  result = strxfrm(d, s, 30);

  printf("%d characters copied: %s\n", result, d);
}
```

```
Output:
16 characters copied: 123456789ABCDEFG
```

21

time.h

The `time.h` header file provides access to the computer system clock, date and time conversion functions, and formatting functions.

Date and time

The `time.h` header file provides access to the computer system clock, date and time conversion functions, and formatting functions.

Three data types are defined in `time.h`: `clock_t`, `time_t`, and `tm`.

The `clock_t` type is a numeric, system dependent type returned by the `clock()` function.

The `time_t` type is a system dependent type used to represent a calendar date and time.

The `struct tm` type contains a field for each part of a calendar date and time.

The `tm` structure members are listed "Tm Structure Members." on page 225.

Table 21.1 Tm Structure Members.

Field	Description	Range min - max
`int tm_sec`	Seconds after the minute	0 - 59
`int tm_min`	Minutes after the hour	0 - 59
`int tm_hour`	Hours after midnight	0 - 23

`int tm_mday`	Day of the month	1 - 31
`int tm_mon`	Months after January	0 - 11
`int tm_year`	Years after 1900	
`int tm_wday`	Days after Sunday	0 - 6
`int tm_yday`	Days after January 1	0 - 365
`int tm_isdst`	Daylight Savings Time flag	

NOTE: IThe `tm_isdst` flag is positive if Daylight Savings Time is in effect, zero if it is not, and negative if such information is not available.

asctime

Description Convert a `tm` structure to a character array.

Prototype

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

Remarks The `asctime()` function converts a `tm` structure, pointed to by `timeptr`, to a character array. The `asctime()` and `ctime()` functions use the same calendar time format. This format, expressed as a `strftime()` format string is `"%a %b %d %H:%M:%S %Y"`.

Return `asctime()`returns a null terminated character array pointer containing the converted `tm structure`.

See Also **`time.h`**: `ctime(), strftime()`

Listing 21.1 Example of asctime() usage.

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

void main(void)
{
  time_t systime;
  struct tm *currtime;

  systime = time(NULL);
  currtime = localtime(&systime);

  puts(asctime(currtime));
}
```

```
Output:
Tue Nov 30 12:56:05 1993
```

clock

Description Return the amount of time the system has been running.

Prototype

```
#include <time.h>
clock_t clock(void);
```

Remarks The `clock()` function returns the amount of time since the computer system was started. To compute the time in seconds, divide the `clock_t` value by `CLOCKS_PER_SEC`, a macro defined in `time.h`.

Return `clock()` returns a `clock_t` type value representing the time since the system was started.

Listing 21.2 Example of clock() usage.

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

void main(void)
{
  clock_t uptime;

  uptime = clock() / CLOCKS_PER_SEC;

  printf("I was booted %ul seconds ago.\n", uptime);

}
```

```
Output:
I was booted 24541 seconds ago.
```

ctime

Description Convert a `time_t` type to a character array.

Prototype

```
#include <time.h>
char *ctime(const time_t *timer);
```

Remarks The `ctime()` function converts a `time_t` type to a character array with the same format used by `asctime()`.

Return `ctime()` returns a null terminated character array pointer containing the converted `time_t` type.

See Also **`time.h`**: `asctime()`, `strftime()`

Listing 21.3 Example of ctime() usage.

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

void main(void)
{
  time_t systime;

  systime = time(NULL);
  puts(ctime(&systime));
}
```

```
Output:
Wed Jul 20 13:32:17 1994
```

difftime

Description Compute the difference between two `time_t` types.

Prototype

```
#include <time.h>
double difftime(time_t t1, time_t t2);
```

Return `difftime()` returns the difference of `t1` minus `t2` expressed in seconds.

Listing 21.4 Example of difftime usage.

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

void main(void)
{
  time_t t1, t2;
  struct tm *currtime;
  double midnight;
```

```
  time(&t1);
  currtime = localtime(&t1);

  currtime->tm_sec = 0;
  currtime->tm_min = 0;
  currtime->tm_hour = 0;
  currtime->tm_mday++;

  t2 = mktime(currtime);

  midnight = difftime(t1, t2);
  printf("There are %f seconds until midnight.\n",
      midnight);
}
```

```
Output:
There are 27892.000000 seconds until midnight.
```

gmtime

Description Convert a `time_t` value to Coordinated Universal Time (UTC), which is the new name for Greenwich Mean Time.

Prototype

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Remarks The `gmtime` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as UTC.

Return The `gmtime()` function returns a pointer to that object.

Listing 21.5 Example of gmtime usage.

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

```
void main(void)
{
  time_t systime;
  struct tm *utc;

  systime = time(NULL);
  utc = gmtime(&systime);

  printf("Universal Coordinated Time:\n");
  puts(asctime(utc));
}
```

```
Output:
Universal Coordinated Time:
Thu Feb 24 18:06:10 1994
```

localtime

Description Convert a `time_t` type to a `struct tm` type.

Prototype

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Remarks The `localtime()` function converts a `time_t` type, pointed to by `timer`, and returns it as a pointer to an internal `struct tm` type. The `struct tm` pointer is `static`; it is overwritten each time `localtime()` is called.

Return `localtime()` converts `timer` and returns a pointer to a `struct tm`.

See Also **`time.h`**: `mktime()`

For Usage Refer to the example for “Example of difftime usage.” on page 229.

mktime

Description Convert a `struct tm` item to a `time_t` type.

Prototype

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Remarks The `mktime()` function converts a `struct tm` type and returns it as a `time_t` type.

The function also adjusts the fields in `timeptr` if necessary. The `tm_sec`, `tm_min`, `tm_hour`, and `tm_day` are processed such that if they are greater than their maximum, the appropriate carry-overs are computed. For example, if `timeptr->tm_min` is 65, `timeptr->tm_hour` will be incremented by 1 and `timeptr->min` will be set to 5.

The function also computes the correct values for `timeptr->tm_wday` and `timeptr->tm_yday`.

Return `mktime()` returns the converted `tm` structure as a `time_t` type.

See Also **`time.h`**`: localtime()`

Listing 21.6 For example of usage

Refer to the example for “Example of difftime usage.” on page 229.

time

Description Return the current system calendar time.

Prototype

```
#include <time.h>
time_t time(time_t *timer);
```

Remarks The `time()` function returns the computer system's calendar time. If `timer` is not a null pointer, the calendar time is also assigned to the item it points to.

Return `time()` returns the system current calendar time.

Listing 21.7 Example of time() usage.

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

void main(void)
{
  time_t systime;
  systime = time(NULL);

  puts(ctime(&systime));
}
```

```
Output:
Tue Nov 30 13:06:47 1993
```

strftime

Description Format a `tm` structure.

Prototype

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
        const char *format,
        const struct tm *timeptr);
```

Remarks The `strftime()` function converts a `tm` structure to a character array using a programmer supplied format.

The `s` argument is a pointer to the array to hold the formatted time.

The `maxsize` argument specifies the maximum length of the formatted character array.

The `timeptr` argument points to a `tm` structure containing the calendar time to convert and format.

NOTE: IThe format argument points to a character array containing normal text and format specifications similar to a `printf()` function format string. Format specifiers are prefixed with a percent sign (%). Doubling the percent sign (%%) will output a single %. Refer to “strftime() conversion characters” on page 234 for a list of format specifiers.

Table 21.2 strftime() conversion characters

Char	Description
`a`	Abbreviated weekday name.
`A`	Full weekday name.
`b`	Abbreviated month name.
`B`	Full month name.
`c`	The strftime() format equaling the format string of `"%x %X"`.
`d`	Day of the month as a decimal number.
`H`	The hour (24-hour clock) as a decimal number from 00 to 23.
`I`	The hour (12-hour clock) as a decimal number from 01 to 12
`j`	The day of the year as a decimal number from 001 to 366
`m`	The month as a decimal number from 01 to 12.

Char	Description
M	The minute as a decimal number from 00 to 59.
p	"AM" or "PM".
S	The second as a decimal number from 00 to 59.
U	The week number of the year as a decimal number from 00 to 52. Sunday is considered the first day of the week.
w	The weekday as a decimal number from 0 to 6. Sunday is (0) zero.
W	The week of the year as a decimal number from 00 to 51. Monday is the first day of the week.
x	The date representation of the current locale.
X	The time representation of the current locale.
y	The last two digits of the year as a decimal number.
Y	The century as a decimal number.
z	The time zone name or nothing if it is unknown.
%	The percent sign is displayed.

Return The `strftime()` function returns the total number of characters in the argument `'s'` if the total number of characters including the null character in the string argument `'s'` is less than the value of `'maxlen'` argument. If it is greater, `strftime()` returns 0.

Listing 21.8 Example of strftime() usage.

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

void main(void)
```

```
{
 time_t lclTime;
 struct tm *now;
 char ts[256];  /* time string */

 lclTime = time(NULL);
 now = localtime(&lclTime);

 strftime(ts, 256,
  "Today's abr.name is %a", now);
 puts(ts);

 strftime(ts, 256,
  "Today's full name is %A", now);
 puts(ts);

 strftime(ts, 256,
  "Today's aabr.month name is %b", now);
 puts(ts);

 strftime(ts, 256,
  "Today's full month name is %B", now);
 puts(ts);

 strftime(ts, 256,
  "Today's date and time is %c",now);
 puts(ts);
 strftime(ts, 256,
"The day of the month is %d", now);
 puts(ts);

 strftime(ts, 256,
"The 24-hour clock hour is %H",now);
 puts(ts);

 strftime(ts, 256,
"The 12-hour clock hour is %H", now);
 puts(ts);

 strftime(ts, 256,
```

```
"Today's day number is %j", now);
 puts(ts);

 strftime(ts, 256,
"Today's month number is %m", now);
 puts(ts);

 strftime(ts, 256,
"The minute is %M", now);
 puts(ts);

 strftime(ts, 256,
"The AM/PM is %p", now);
 puts(ts);

 strftime(ts, 256,
"The second is %S", now);
 puts(ts);

 strftime(ts, 256,
"The week number of the year,\
starting on a Sunday is %U", now);
 puts(ts);

 strftime(ts, 256,
"The number of the week is %w", now);
 puts(ts);

 strftime(ts, 256, "The week number of the year,\
starting on a Monday is %W", now);
 puts(ts);

 strftime(ts, 256, "The date is %x", now);
 puts(ts);

 strftime(ts, 256, "The time is %X", now);
 puts(ts);

 strftime(ts, 256,
  "The last two digits of the year are %y", now);
```

```
 puts(ts);

 strftime(ts, 256, "The year is %Y", now);
 puts(ts);

 strftime(ts, 256, "%Z", now);
 if (strlen(ts) == 0)
  printf("The time zone cannot be determined\n");
 else
  printf("The time zone is %s\n", ts);
}
```

```
Results
Today's abr.name is Thu
Today's full name is Thursday
Today's aabr.month name is Aug
Today's full month name is August
Today's date and time is Aug 24 11:42:16 1995
The day of the month is 24
The 24-hour clock hour is 11
The 12-hour clock hour is 11
Today's day number is 236
Today's month number is 08
The minute is 42
The AM/PM is AM
The second is 16
The week number of the year,starting on a Sunday is 34
The number of the week is 4
The week number of the year,starting on a Monday is 34
The date is Aug 24 1995
The time is 11:42:16
The last two digits of the year are 95
The year is 1995
The time zone cannot be determined
```

22

unistd.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

chdir

Description Change the current directory.

Prototype

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

Remarks The function `chdir()` is used to change from one directory to a different directory or folder. Example of usage is given in "Example of chdir() usage." on page 240

Return `chdir()` returns zero, if successful. If unsuccessful `chdir()` returns negative one and sets `errno`.

See Also `<errno.h>`

Listing 22.1 Example of chdir() usage.

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

#define SIZE FILENAME_MAX
#define READ_OR_WRITE0x0 /* fake a UNIX mode */

void main(void)
{
char folder[SIZE];
char curFolder[SIZE];
char newFolder[SIZE];
int folderExisted = 0;

  /* Get the name of the current folder or directory */
  getcwd( folder, SIZE );
  printf("The current Folder is: %s", folder);

  /*  create a new sub folder */
  /* note mode parameter ignored on Mac */
  sprintf(newFolder,"%s%s", folder, "Sub" );
  if( mkdir(newFolder, READ_OR_WRITE ) == -1 )
  {
    printf("\nFailed to Create folder");
    folderExisted = 1;
  }

  /* change to new folder */
  if( chdir( newFolder) )
  {
    puts("\nCannot change to new folder");
    exit(EXIT_FAILURE);
  }

  /* show the new folder or folder  */
  getcwd( curFolder, SIZE );
  printf("\nThe current folder is: %s", curFolder);
```

```
  /* go back to previous folder */
  if( chdir(folder) )
  {
    puts("\nCannot change to old folder");
    exit(EXIT_FAILURE);
  }

  /* show the new folder or folder  */
  getcwd( curFolder, SIZE );
  printf("\nThe current folder is again: %s", curFolder);

  if (!folderExisted)
  {
  /* remove newly created directory  */
  if (rmdir(newFolder))
    {
    puts("\nCannot remove new folder");
    exit(EXIT_FAILURE);
    }
    else
    puts("\nNew folder removed");

  /* attempt to move to non-existant directory  */
    if (chdir(newFolder))
   puts("Cannot move to non-existant folder");
  }
  else  puts("\nPre-existing folder not removed");

}
```

```
Output
The current Folder is: Macintosh HD:C Reference:
The current folder is: Macintosh HD:C Reference:Sub:
The current folder is again: Macintosh HD:C Reference:
New folder removed
Cannot move to non-existant folder
```

close

Description Close an open file.

Prototype

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

Remarks The `close()` function closes the file specified by the argument. This argument is the value returned by `open()`. Example of usage is given in "Example of close() usage." on page 242

Return If successful, `close()` returns zero. If unsuccessful, `close()` returns negative one and sets `errno`.

See Also `fcntl.h:open(), stdio.h:fclose(), <errno.h>`

Listing 22.2 Example of close() usage.

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


#define SIZE FILENAME_MAX
#define MAX 1024

char fname[SIZE] = "DonQ.txt";


void main(void)
{
  int fdes;
  char temp[MAX];
  char *Don = "In a certain corner of la Mancha, the name
of\n\
which I do not choose to remember,...";
```

```
 char *Quixote = "there lived\none of those country\
gentlemen, who adorn their\nhalls with rusty lance\
and worm-eaten targets.";

 /* NULL terminate temp array for printf */
 memset(temp, '\0', MAX);

 /* open a file */
 if((fdes = open(fname,O_RDWR|O_TRUNC|O_BINARY ))==1)
 {
   printf("Can not open %s", fname);
   exit( EXIT_FAILURE);
 }

 /* write to a file  */
 if( write(fdes, Don, strlen(Don)) == -1)
 {
   printf("Write Error");
   exit( EXIT_FAILURE );
 }

 /*move back to over write ... characters */
 if( lseek( fdes, -3L, SEEK_CUR ) == -1L)
 {
   printf("Seek Error");
   exit( EXIT_FAILURE );
 }

 /* write to a file  */
 if( write(fdes, Quixote, strlen(Quixote)) == -1)
 {
   printf("Write Error");
   exit( EXIT_FAILURE );
 }

 /* move to beginning of file for read */
 if( lseek( fdes, 0L, SEEK_SET ) == -1L)
 {
   printf("Seek Error");
   exit( EXIT_FAILURE );
```

```
  }

  /* read the file  */
  if( read( fdes, temp, MAX ) == 0)
  {
    printf("Read Error");
    exit( EXIT_FAILURE);
  }

  /* close the file */
  if(close(fdes))
  {
    printf("File Closing Error");
    exit( EXIT_FAILURE );
  }

   puts(temp);

}
```

```
Result
In a certain corner of la Mancha, the name of
which I do not choose to remember,there lived
one of those country gentlemen, who adorn their
halls with rusty lance  and worm-eaten targets.
```

cuserid

Description Retrieve the current user's ID.

Prototype

```
#include <unistd.h>
char *cuserid(char *string);
```

Remarks The function `cuserid()` returns the user name associated with the current process. If the string argument is `NULL`, the file name is stored in an internal buffer. If it is not `NULL`, it must be at least

FILENAME_MAX large. Example of usage is given in “Example of cuserid() usage.” on page 245

Return cuserid() returns a character pointer to the current user’s ID.

Listing 22.3 Example of cuserid() usage.

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

void main(void)
{
 char *c_id = NULL;
 printf("The current user ID is %s\n",cuserid(c_id));
}
```

```
Result
The current user ID is Metrowerks
```

exec

Description Load and execute a child process within the current program memory.

Prototype

```
#include <unistd.h>
int exec(const char *path, ...);
```

NOTE: ITAll exec-type calls pass through exec(), because argument passing (argc, argv) doesn't exist for GUI applications

Table 22.1 The exec() type functions

UNIX Function	On the Macintosh System
#define execl	exec
#define execv	exec
#define execle	exec
#define execve	exec
#define execlp	exec
#define execvp	exec

DescriptionLaunches the application named and then quits upon successful launch. Example of usage is given in "Example of exec() usage." on page 246

DescriptionIf successful `exec()` returns zero. If unsuccessful `exec()` returns negative one and sets `errno` according to the error.

See Also `SIOUX.h, errno()`

Listing 22.4 Example of exec() usage.

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


#define SIZE FILENAME_MAX
char appName[SIZE] = "Macintosh HD:SimpleText";

void main(void)
{
  SIOUXSettings.autocloseonquit = 1;
  SIOUXSettings.asktosaveonclose = 1;
```

```
  printf("Original Program\n");
  exec(appName);
  printf("program terminated"); /* not displayed */
}
```

```
result
Display "Original Program"
after the close of the program
the SimpleText application is launched
```

execl

Remark All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

execle

Remark All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

execlp

Description All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

execv

Description All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

execve

Description All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

excevp

Description All “exec”-type calls pass through `exec()`, because argument passing (`argc`, `argv`) doesn't exist for Mac application

getcwd

Description Get the current directory.

Prototype

```
#include <unistd.h>
char *getcwd(char *buf, int size);
```

Remarks The function `getcwd()` takes two arguments. One is a buffer large enough to store the full directory pathname, the other argument is the size of that buffer.

Return If successful, `getcwd()` returns a pointer to the buffer. If unsuccessful, `getcwd()` returns NULL and sets `errno`.

See Also `<errno.h>`

Listing 22.5 For example of getcwd usage

Refer to “Example of chdir() usage.” on page 240.

getlogin

Description Retrieve the username that started the process.

Prototype

```
#include <unistd.h>
char *getlogin(void);
```

Remarks The function `getlogin()` retrieves the name of the user who started the program. Example of usage is given in "Example of getlogin() usage." on page 249

NOTE: IThe Mac doesn't have a login, so this function returns the Owner Name from the Sharing Setup Control Panel

Return `getlogin()` returns a character pointer.

Listing 22.6 Example of getlogin() usage.

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

void main(void)
{
  printf("The login name is %s\n", getlogin());
}
```

```
result
The login name is Metrowerks
```

getpid

Description Retrieve the process identification number.

Prototype `#include <unistd.h>`

Table 22.2 getpid() Macros

Macro	Return Value	Represents
#define getpid()	((int) 9000)	Process ID
#define getppid()	((int) 8000)	Parent process ID
#define getuid()	((int) 200)	Real user ID
#define geteuid()	((int) 200)	Effective user ID
#define getgid()	((int) 100)	Real group ID
#define getegid()	((int) 100)	Effective group ID
#define getpgrp()	((int) 9000)	Process group ID

Remarks The `getpid()` function returns the unique number (`Process ID`) for the calling process. Example of usage is given in "Example of getpid() usage." on page 250

Return `getpid()` returns an integer value.

NOTE: IThese various related `getpid()` type functions don't really have any meaning on the Mac. The values returned are those that would make sense for a typical user process under UNIX.

Return `getpid()` always returns a value. There is no error returned.

Listing 22.7 Example of getpid() usage.

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

void main(void)
{
  printf("The process ID is %d\n", getpid());
```

```
}
```

```
Result
The process ID is 9000
```

isatty

Description Determine a specified file_id

Prototype

```
#include <unistd.h>
int isatty(int fildes);
```

Remarks The function `isatty()` determines if a specified `file_id` is attached to the console, or if re-direction is in effect. Example of usage is given in "Example of isatty() ttyname() usage." on page 251

Return `isatty()` returns a non-zero value if the file is attached to the console. It returns zero if Input/Output redirection is in effect.

See Also `console.h:ccommand()`

Listing 22.8 Example of isatty() ttyname() usage.

```
#include <console.h>
#include <stdio.h>
#include <unistd.h>
#include <unix.h>

void main(int argc, char **argv)
{
  int i;
  int file_id;
   argc = ccommand(&argv);

   file_id = isatty(fileno(stdout));
   if(!file_id )
```

```
   {
  for (i=0; i < argc; i++)
    printf("command line argument [%d] = %s\n",
          i, argv[i]);
  }
  else printf("Output to window");

  printf("The associated terminal is %s",
        ttyname(file_id) );
}
```

```
Result if file redirection is chosen using the command line
arguments
Metrowerks CodeWarrior.
Written to file selected:

command line argument [0] = CRef
command line argument [1] = Metrowerks
command line argument [2] = CodeWarrior
The associated terminal is SIOUX
```

lseek

Description Seek a position on a file stream.

Prototype

```
#include <unistd.h>
long lseek(int fildes, long offset, int origin);
```

Remarks The function `lseek()` sets the file position location a specified byte offset from a specified initial location.

NOTE: IThe origin of the offset must be one of three positions:

```
SEEK_SET        Beginning of file
SEEK_CUR        Current Position
SEEK_END        End of File
```

Return If successful, `lseek()` returns the number of bytes offset. If unsuccessful, it returns a value of negative one long integer.

See Also `stdio.h:seek(), stdio.h:tell(), fcntl.h:open()`

Listing 22.9 For example of lseek() usage

Refer to “Example of close() usage.” on page 242.

read

Description Read from a file stream that has been opened for unformatted Input/Output.

Prototype

```
#include <unistd.h>
int read(int fildes, char *buf, int count);
```

Remarks The function `read()` copies the number of bytes specified by the count argument, from the file to the character buffer. File reading begins at the current position. The position moves to the end of the read position when the operation is completed.

NOTE: IThis function should be used in conjunction with **`unistd.h:`**`write()`, and **`fcntl.h:`**`open()` only.

Return `read()` returns the number of bytes actually read from the file. In case of an error a value of negative one is returned and errno is set.

See Also `stdio.h:fread(), fcntl.h:open()`

Listing 22.10 For example of read() usage

```
Refer to “Example of close() usage.” on page 242.
```

rmdir

Description Delete a directory or folder.

Prototype

```
#include <unistd.h>
int rmdir(const char *path);
```

Remarks The `rmdir()` function removes the directory (folder) specified by the argument.

Return If successful, `rmdir()` returns zero. If unsuccessful, `rmdir()` returns negative one and sets `errno`.

See Also `<errno.h>`

Listing 22.11 For example of rmdir() usage

```
Refer to“Example of chdir() usage.” on page 240.
```

sleep

Description Delay program execution for a specified number of seconds.

Prototype

```
#include <unistd.h>
unsigned int sleep(unsigned int sleep);
```

Remarks The function `sleep()` delays execution of a program for the time indicated by the unsigned integer argument. For the Macintosh sys-

tem there is no error value returned. Example of usage is given in "Example of sleep() usage." on page 255

Return `sleep()` returns zero.

Listing 22.12 Example of sleep() usage.

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

void main(void)
{


  printf("Output to window\n");
  fflush(stdout); /* needed to force output */

  sleep(3);

  printf("Second output to window");
}
```

```
Result
Output to window
< there is a delay now >
Second output to window
```

ttyname

Description Retrieve the name of the terminal associated with a file ID.

Prototype

```
#include <unistd.h>
char *ttyname(int fildes);
```

Remarks The function `ttyname()` retrieves the name of the terminal associated with the file ID.

Return `ttyname()` returns a character pointer to the name of the terminal associated with the file ID, or NULL if the file ID doesn't specify a terminal.

Listing 22.13 For example of ttyname() usage

```
Refer to “Example of isatty() ttyname() usage.” on page 251.
```

unlink

Description Delete (unlink) a file.

Prototype

```
#include <unistd.h>
int unlink(const char *path);
```

Remarks The function `unlink()` removes the specified file from the directory. Example of usage is given in “Example of unlink() usage.” on page 256

Return If successful, `unlink()` returns zero. If unsuccessful, it returns a negative one.

Listing 22.14 Example of unlink() usage.

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

#define SIZE FILENAME_MAX

void main(void)
{
  FILE *fp;
  char fname[SIZE] = "Test.txt";

  /* create a file */
```

```
  if( (fp =fopen( fname,"w") ) == NULL )
  {
    printf("Can not open %s for writing", fname);
    exit( EXIT_FAILURE);
  }
  else printf("%s was opened for writing\n",fname);

  /* display it is available */
  if( !fclose(fp) ) printf("%s was closed\n",fname);

  /* delete file */
  if( unlink(fname) )
  {
    printf("%s was not deleted",fname);
    exit( EXIT_FAILURE );
  }

  /* show it can't be re-opened */
  if( (fp =fopen( fname,"r") ) == NULL )
  {
    printf("Can not open %s for reading it was deleted",
         fname);
    exit( EXIT_FAILURE);
  }
  else printf("%s was opened for reading\n",fname);

}
```

```
Result
Test.txt was opened for writing
Test.txt was closed
Can not open Test.txt for reading it was deleted
```

write

Description Write to an un-formatted file stream.

Prototype

```
#include <unistd.h>
int write(int fildes, const char *buf, int count);
```

Remarks The function `write()` copies the number of bytes in the `count` argument from the character array buffer to the file `fildes`. The file position is then incremented by the number of bytes written.

NOTE: IThis function should be used in conjunction with **`unistd.h:`**`read()`, and **`fcntl.h:`**`open()` only.

Return `write()` returns the number of bytes actually written.

See Also `stdio.h:fwrite(), unistd.h:read(), fcntl.h:open()`

Listing 22.15 For example of write() usage

Refer to “Example of close() usage.” on page 242.

23

unix.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

fdopen

Description Converts a file descriptor to a stream.

Prototype

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

Remarks This function creates a stream for the file descriptor `fildes`. You can use the stream with such standard I/O functions as `fprintf()` and `getchar()`. In Metrowerks C/C++, it ignores the value of the `mode` argument.

Return If it is successful, `fdopen()` returns a stream. If it encounters an error, `fdopen()` returns `NULL`.

See Also `fileno()`, **`fcntl.h`**`:open()`

Listing 23.1 Example of fdopen() usage.

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

void main(void)
{
  int fd;
  FILE *str;

  fd = open("mytest", O_WRONLY | O_CREAT);

  /* Write to the file descriptor */
  write(fd, "Hello world!\n", 13);
  /* Convert the file descriptor to a  stream */

  str = fdopen(fd,"w");

  /* Write to the stream */
  fprintf(str, "My name is %s.\n", getlogin());

  /* Close the stream. */
  fclose(str);
  /* Close the file descriptor */
  close(fd);
}
```

fileno

Description Converts a stream to a file descriptor

Prototype

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

Remarks This function creates a file descriptor for the stream `stream`. You can use the file descriptor with other functions in `unix.h`, such as `read()` and `write()`.

For the standard I/O streams `stdin`, `stdout`, and `stderr`, `fileno()` returns the following values:

Table 23.1 File Descriptors for the Standard I/O Streams.

This function call…	Returns this file descriptor…
`fileno(stdin)`	0
`fileno(stdout)`	1
`fileno(stderr)`	2

Return If it is successful, `fileno()` returns a file descriptor. If it encounters an error, it returns `-1` and sets `errno`.

See Also `fdopen(), open()`

Figure 23.1 Example of fdopen() usage.

```
#include <unix.h>

void main(void)
{
  write(fileno(stdout), "Hello world!\n", 13);
}
```

```
Reult
Access time:       Tue Apr 18 22:28:22 1995
Modification time: Tue Apr 18 22:28:22 1995
Creation time:     Tue Apr 18 11:28:41 1995
Block size:        11264
Number of blocks:  1
```

tell

Description Returns the current offset for a file.

Prototype

```
#include <unix.h>
long tell(int fildes);
```

Remarks This function returns the current offset for the file associated with the file descriptor fildes. The value is the number of bytes from the file's beginning.

Return If it is successful, `tell()` returns the offset. If it encounters an error, `tell()` returns `-1L`

See Also **`stdio.h`**: `ftell()`; **`unix.h`**: `lseek()`

Listing 23.2 Example of read() usage.

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

void main(void)
{
  int fd;
  long int pos;

  fd = open("mytest", O_RDWR | O_CREAT | O_TRUNC);
  write(fd, "Hello world!\n", 13);
  write(fd, "How are you doing?\n", 19);

  pos = tell(fd);

  printf("You're at position %ld.", pos);

  close(fd);
}
```

```
Result
This program prints the following to standard output:
You're at position 32.
```

24

utime.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

utime

Purpose Sets a file's modification time.

Prototype

```
#include <utime.h>
int utime(const char *path,
    const struct utimbuf *buf);
```

Remarks This function sets the modification time for the file specified in `path`. Since the Macintosh does not have anything that corresponds to a file's access time, it ignores the `actime` field in the `utimbuf` structure.

If `buf` is `NULL`, `utime()` sets the modification time to the current time. If `buf` points to a `utimbuf` structure, `utime()` sets the modification time to the time specified in the `modtime` field of the structure.

The utimbuf structures contains the fields in “The utimbuf structure” on page 266.

Table 24.1 The utimbuf structure

This field…		is the…
time_t	actime	Access time for the file. Since the Macintosh has nothing that corresponds to this, `utime()` ignores this field.
time_t	modtime	The last time this file was modified.

Return

If it is successful, `utime()` returns zero. If it encounters an error, `utime()` returns `-1` and sets `errno`.

See Also

`time.h`: `ctime(), mktime();` **`unix.h`**: `utimes(), stat(), fstat()`

Listing 24.1 Example for utime()

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

void main(void)
{
  struct utimbuf timebuf;
  struct tm date;
  struct stat info;


  /* Create a calendar time for
  Midnight, Apr. 4, 1964. */
  date.tm_sec=0;    /* Zero seconds   */
  date.tm_min=0; /* Zero minutes   */
  date.tm_hour=0;/* Zero hours     */
  date.tm_mday=4;   /* 4th day        */
  date.tm_mon=3;/* .. of April    */
  date.tm_year=64; /* ...in 1964     */
```

```
  date.tm_isdst=-1;/* Not daylight savings */
  timebuf.modtime=mktime(&date);
    /* Convert to calendar time.    */

  /* Change modification date to  *
   * midnight, Apr. 4, 1964.   */
  utime("mytest", &timebuf);
  stat("mytest", &info);
  printf("Mod date is %s", ctime(&info.st_mtime));

  /* Change modification date to */
   * right now.                   */
  utime("mytest", NULL);
  stat("mytest", &info);
  printf("Mod date is %s", ctime(&info.st_mtime));
}
```

```
This program might print the following to standard output:
Mod date is Sat Apr  4 00:00:00 1964
Mod date is Mon Apr 10 20:43:09 1995
```

utimes

Description Sets a file's modification time

Prototype

```
#include <utime.h>
int utimes(const char *path,
    struct timeval buf[2]);
```

Remarks This function sets the modification time for the file specified in path to the second element of the array `buf`. Each element of the array `buf` is a `timeval` structure, which has the fields in "The utimbuf structure" on page 266.

Table 24.2 The timeval structure

This field		is the
int t	tv_sec	Seconds
int	tv_usec	Microseconds. Since the Macintosh does not use microseconds, `utimes()` ignores this.

The first element of `buf` is the access time. Since the Macintosh does not have anything that corresponds to a file's access time, it ignores that element of the array.

Return

If it is successful, `utimes()` returns `0`. If it encounters an error, `utimes()` returns `-1` and sets `errno`.

See Also

`time.h`: `ctime(), mktime();` **`unix.h`**: `utime(), fstat(), stat()`

Listing 24.2 Example for utimes()

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

void main(void)
{
  struct tm date;
  struct timeval buf[2];
  struct stat info;


/* Create a calendar time for
Midnight, Sept. 9, 1965.*/
  date.tm_sec=0;      /* Zero seconds */
  date.tm_min=0;      /* Zero minutes */
  date.tm_hour=0;     /* Zero hours */
  date.tm_mday=9;     /* 9th day */
  date.tm_mon=8;      /* .. of September  */
```

```
  date.tm_year=65;    /* ...in 1965   */
  date.tm_isdst=-1;   /* Not daylight savings  */
  buf[1].tv_sec=mktime(&date);
  /* Convert to calendar time.  */

  /* Change modification date to   *
   * midnight, Sept. 9, 1965.      */
  utimes("mytest", buf);
  stat("mytest", &info);
  printf("Mod date is %s", ctime(&info.st_mtime));
}
```

```
This program prints the following to standard output:
Mod date is Thu Sep  9 00:00:00 1965
```

25

utsname.h

The header file `unix.h` contains several functions that are useful for porting a program from UNIX.

UNIX Compatibility

The header file `unix.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the Macintosh Toolbox.

NOTE: IIf you're porting a UNIX or DOS program, you might also need the functions in `console.h` and `SIOUX.h`.

uname

Description Gets information about the Macintosh you're using.

Prototype

```
#include <utsname.h>
int uname(struct utsname *name);
```

Remarks This function gets information on the Macintosh you're using and puts the information in the structure that `name` points to. The structure contains the fields listed in "The utsname structure" on page 272. All the fields are null-terminated strings.

Table 25.1 The utsname structure

This field…	is…
sysnam	`"MacOS"`.
nodename	The name of the Macintosh entered in the field Macintosh Name in the Sharing Setup control panel.
release	The major release number of the system software version. For example, `"7"` is for System 7.
version	The minor release numbers of the system software version. For example, if `release` is `"7"` and `version` is `"51"`, you're using System 7.5.1.
machine	The type of the Macintosh you're using.

Return If it is successful, `uname()` returns zero. If it encounters an error, `uname()` returns `-1` and sets `errno`.

See Also **`unix.h`**: `fstat(), stat()`

Listing 25.1 Example of uname() usage.

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

void main(void)
{
  struct utsname name;

  uname(&name);

  printf("Operating System: %s\n", name.sysname);
  printf("Node Name:        %s\n", name.nodename);
  printf("Release:          %s\n", name.release);
```

```
  printf("Version:          %s\n", name.version);
  printf("Machine:          %s\n", name.machine);
}
```

```
This application could print the following:
Operating System: MacOS
Node Name:        Chan's PowerMac
Release:          7
Version:          51
Machine:            Power Macintosh
```

This machine is a Power Macintosh running Version 7.5.1 of the MacOS. The Macintosh Name field of the Sharing Setup control panel contains "Chan's PowerMac"

Index

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

CodeWarrior

MSL C Reference

Credits

writing lead:	Ron Liechty
other writers:	Marc Paquette
engineering:	Michael Marcotty, Benjamin Koznik , Andreas Hommel
frontline warriors:	Karine Gabrini, & all CodeWarriors

Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	QuickStart Guide
Getting started using CodeWarrior	QuickStart Guide; Tutorials (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	IDE User's Guide
Debugging	Debugger Manual
Important last-minute information on new features and changes	Release Notes folder
Creating Macintosh and Power Macintosh software	Targeting Mac OS; Mac OS folder
Creating Microsoft Win32/x86 software	Targeting Win32; Win32/x86 folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	Targeting Magic Cap; Magic Cap folder
Using ToolServer with the CodeWarrior editor	IDE User's Guide
Controlling CodeWarrior through AppleScript	IDE User's Guide
Using CodeWarrior to program in MPW	Command Line Tools Manual
C, C++, or 68K assembly-language programming	C, C++, and Assembly Reference; MSL C Reference; MSL C++ Reference
Pascal or Object Pascal programming	Pascal Language Manual; Pascal Library Reference
Fixing compiler and linker errors	Errors Reference
Fixing memory bugs	ZoneRanger Manual
Speeding up your programs	Profiler Manual
PowerPlant	The PowerPlant Book; PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	Constructor Manual
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	Discover Programming for Macintosh
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	Quick Start Guide
Contacting Metrowerks about problems and suggestions using CodeWarrior software	email Report Forms in the Release Notes folder
Sample programs and examples	CodeWarrior Examples folder; The PowerPlant Book; PowerPlant Advanced Topics; Tutorials (Apple Guide)
Problems other CodeWarrior users have solved	Internet newsgroup [docs] folder

Revision code 1112rdl