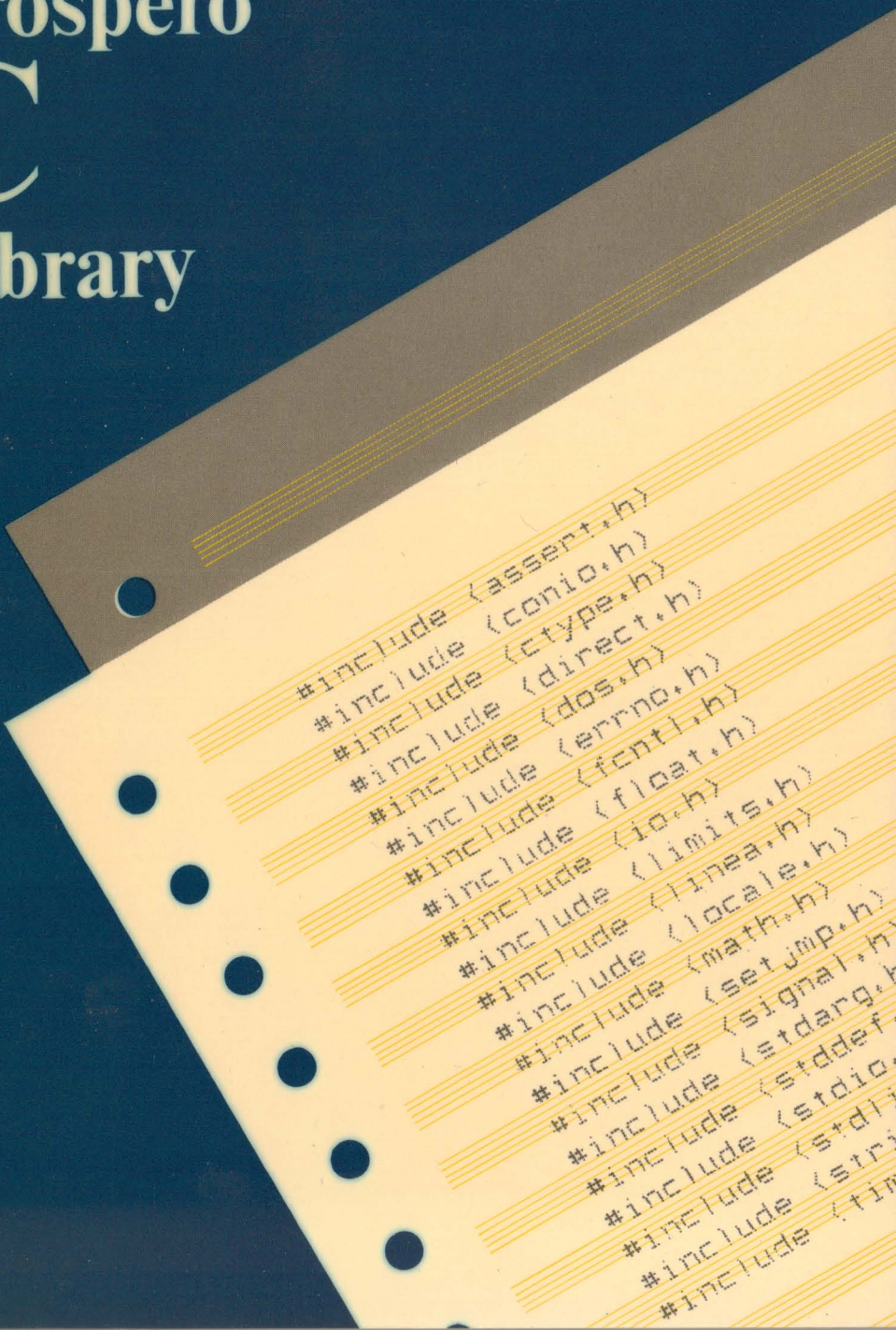
# Prospero C Library



# Prospero C Library

September 1990

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

## COPYRIGHT



## DISCLAIMER

While every effort is made to ensure accuracy, Prospero Software cannot be held responsible for errors or omissions, and reserve the right to revise this document without notice.

## TRADEMARKS

Acknowledgement is made for references in this manual to Apple, Lisa and Macintosh, which are trademarks of Apple Computer Inc., to WordStar, which is a trademark of MicroPro International Corp., to Digital Research and GEM, which are trademarks of Digital Research Inc., to Atari and Atari ST, which are trademarks of Atari Corp., to Motorola and MC68000, which are trademarks of Motorola Inc., and to Unix, which is a trademark of AT&T Bell Laboratories.

Prospero C, Pro Fortran-77, Prospero Fortran, Pro Pascal and Prospero Pascal are trademarks of Prospero Software.

Prospero Software, Inc.
100 Commercial Street, Suite 306
Portland, Maine 04101
U.S.A

Prospero Software Ltd.
190 Castelnau
London SW13 9DH
England

# TABLE OF CONTENTS

# 1 INTRODUCTION

This volume describes the implementation of the Prospero C Library, including a detailed description of each library function which is available to be called from Prospero C programs.

Section 2 gives a breakdown of the library by header file. Each header file defines several functions that are related in usage. Most of these are part of the ANSI standard, but may contain functions that are not. Other header files have been added by Prospero. Section 2 describes each header file, the functions and macros it defines and how they relate with each other.

Section 3 gives a detailed description of each library function in turn, detailing its parameters, return value and purpose, as well as giving a short example of how it might be used. The introduction to Section 3 gives many details common to many of the functions, including the usage of functions defined as macros, and dealing with errors.

Section 4 gives an index of functions, specifying the header file that needs to be included for each function.

This manual does not cover the AES and VDI bindings, which are treated separately in volumes III and IV.

# 2 LIBRARY HEADER FILES

This section gives an overview of the Prospero C Library organized by purpose, by describing the contents of each library header file. Each section includes a list of all the functions whose prototypes are found in the header file, and a brief description of what sort of thing these functions are used for. They also give general information which is relevant to all or most of the functions listed, and indicate how the functions inter-relate. Often, there is a short example program to illustrate the use of some of the functions to show how they fit together.

In order to use any of the functions described below, the programmer should ensure that the relevant header file is included, using a statement of the form

```
#include <header.h>
```

The angle brackets around the header file name indicate that the compiler should look in the directory nominated for header files (see Volume I, Part I for how this is done). Enclosing a filename in double quotes causes the compiler to search for the given filename relative to the current directory, and would be used to include C code if a large source is divided into several smaller files for easier editing, for example.

It is normal practice to include all header files at the start of the program, before any declarations or functions. It does not matter in what order the header files are included, nor if any file is included more than once.

The header files `limits.h` and `float.h`, which define macros giving details of the ranges of arithmetic types, are described in Volume I, appendix G.

The header files `aesbind.h` and `vdibind.h`, which define the AES and VDI bindings, are described in volumes III and IV.

## 2.1 Diagnostics <assert.h>

The assert.h header file defines the assert macro. This is used to insert diagnostics into a program in such a way that they can be easily disabled to create a faster and smaller program when it is believed to be free of bugs. The way in which it is used is described in detail under assert, in section 3.

## 2.2 Console input/output <conio.h>

The conio.h header file defines the following functions concerned with console input and output.

| | |
|---|---|
| getch | get character from keyboard |
| getche | get character form keyboard, with echo |
| ungetch | unget character from keyboard |
| kbhit | test if key pressed |
| putch | put character to screen |

These functions are used to provide interaction with the console. They only get or put single characters; for more powerful print and input functions refer to stdio.h. Note that getting a character from the keyboard is not equivalent to getting one from standard input, nor is putting one to the screen equivalent to putting one to standard output. Both standard input and standard output may be redirected to refer to files rather than to the console – the console input/output functions will not be affected by this. Also, unless the buffering mode of standard input is changed, getting characters from standard input will cause the program to wait until a whole line has been typed, then return the characters in the line one by one. Getting characters from the keyboard using one of the functions in conio.h will return each character as it is typed.

None of the functions defined in conio.h are part of the draft ANSI standard.

e.g.,

```
/* function that reads all the characters from the */
/* buffer to leave it empty                        */

#include <conio.h>

void empty_buffer(void)
{  while (kbhit())
     getch();
}
```

## 2.3 Character handling <ctype.h>

The ctype.h header file defines the following functions:

| | |
|---|---|
| isalnum | tests if a character is alphanumeric |
| isalpha | tests if a character is a letter |
| isascii | tests if a character is an ascii character |
| iscntrl | tests if a character is a control character |
| isdigit | tests if a character is a digit |
| isgraph | tests if a character is a non-space printing character |
| islower | tests if a character is a lower case letter |
| isprint | tests if a character is a printable character |
| ispunct | tests if a character is a punctuation character |
| isspace | tests if a character is a white-space character |
| isupper | tests if a character is an upper case letter |
| isxdigit | tests if a character is a hex digit |
| toascii | converts a character to ascii character |
| tolower | converts a letter to lower case |
| toupper | converts a letter to upper case |

The functions whose names begin is... are used to test whether a character falls into a particular category, returning non-zero if the character does fall in the appropriate category, otherwise zero. Those beginning to... map characters to a particular range of characters. These functions are described in detail in section 3.

Most of the functions in ctype.h are in fact defined as macros.

e.g.,

```
#include <ctype.h>

char *s;

while (*s)
  if isprint(*s)
    putchar (*s++);
  else
  { putchar('.');
    s++;
  };
```

## 2.4 Directory handling <direct.h>

The direct.h header file defines the following functions for manipulating disks and directories.

| | |
|---|---|
| chdir | change directory |
| drivemap | get bitmap of drives that are connected |
| getcwd | get current pathname |
| getdfs | get disk information |
| getdisk | find disk drive |
| mkdir | make directory |
| rmdir | remove directory |
| setdisk | change disk drive |

These functions are rather operating system specific, and therefore not part of the draft ANSI C standard. Directories can be added and removed, the current disk drive and directory can be changed (a filename without a drive or path specifier is assumed to refer to the current drive and directory). The disk information returned by getdfs enables the size of the disk and the amount of free space to be calculated. Disk drives that are available can be found using drivemap, which is useful as functions such as setdisk do not indicate whether the request is valid.

Note that chdir, rmdir and mkdir all allow a forward slash character to be used in place of a backslash when specifying path names, as entering backslashes in string literals requires two backslashes to be typed.

e.g.,

```
#include <direct.h>

main()

{   struct DISKINFO dinfo;
    int i;
    unsigned map;
    long free;

    map = drivemap();  /* Get bitmap of drives */
    printf("Report on Disk Drives:-\n\n");
    for(i=0;i<16;i++)
        if (map & (1 << i))
        {  getdfs (i+1, &dinfo);
           free = dinfo.free * dinfo.bps * dinfo.spc;
           printf ("Drive %c has %ld bytes free\n",
                       i+'A', free);
        }
}
```

## 2.5 Operating system functions <dos.h>

The dos.h header file defines the following functions to call the Atari's operating system:

| | |
|---|---|
| gemdos | make a GEMDOS call |
| bios | make a BIOS call |
| xbios | make an XBIOS call |

There are also a large number of macros expanding into calls of gemdos which can be used for making calls to specific operating system functions. Each of the above functions requires a variable number of parameters, depending on which specific GEMDOS, XBIOS or BIOS function is required. The first parameter is always an int, and specifies the function number. The number, meaning and type of the other parameters depends on the value of this first parameter – refer to Atari technical specifications for details of the functions available. Most of the more important ones are duplicated by Prospero C library routines.

None of these functions are part of the draft ANSI standard.

e.g.,

```
#include <dos.h>

/* function to set time and date as returned by   */
/* mktime function                                */
void set_time(long date_time)
{  gemdos (0x2b, (int) ( (date_time >> 16) & 0xffff) );
   gemdos (0x2d, (int) (date_time & 0xffff) );
}

/* function to set the caps lock on or off        */
void caps_lock(int status)
{   if (status)
       bios(11, (int) (bios (11, -1) | 16) );
    else
       bios(11, (int) (bios (11, -1) & ~16) );
}

/* function to set the keyboard delay and repeat */
/* rate, but not allowing silly values           */
void keyboard_rate(int delay, int rate)
{   xbios (35,
          (delay < 10) ? 10 : delay,
          (rate  < 10) ? 10 : rate      )
}
```

## 2.6 Error numbers <errno.h>

The errno.h header file defines the macro errno, which indicates the most recent error detected by the Prospero C library, as well as macros for every error number which may be placed in errno by the library. The usage of errno and the treatment of errors by the library is described further in section 3.5.2.

Many of the error number macros defined in errno.h originate from the operating system or bios. Error codes returned by the gemdos and bios functions are always negative, but are negated before being placed in errno by the library so the all the error codes are positive.

The functions strerror and perror can be used to convert an error number to a meaningful error message.

e.g.,

```
#include <errno.h>

main()

{ FILE *stream  = fopen("myfile", "w");
  if (stream == NULL)
  { if (errno == EINVDRV)
    { /* bad drive. Ask for new drive and make */
      /* that current drive                    */
      ...
    }
  else
    perror("Unable to open myfile");
  }
  ...
}
```

## 2.7 File control macros <fcntl.h>

The fcntl.h header file defines the following macros that may be used when opening an unbuffered file (see section 2.8):

| | |
|---|---|
| O_APPEND | write starting at current end of file |
| O_CREAT | create a file if it does not exist |
| O_EXCL | if a file is to be created it must not already exist |
| O_NDELAY | not used - included only for compatibility |
| O_RDWR | read or write |
| O_RDONLY | read only |
| O_TEXT | open file in text mode |
| O_TRUNC | truncate file to zero length if it already exists |
| O_WRONLY | write only |
| | |
| S_IREAD | read permission attribute |
| S_IWRITE | write permission attribute |
| S_SUBDIR | subdirectory attribute |

This header file is not part of the draft ANSI standard.

## 2.8 Unbuffered input/output <io.h>

The io.h header file defines the following functions connected with the use of unbuffered, operating system level file handling:

| | |
|---|---|
| access | access status of file |
| chmod | change file status |
| close | close file |
| creat | create file |
| dup | duplicate handle |
| dup2 | change handle |
| eof | test if end of file |
| filelength | find length of file |
| findfirst | find first file with name matching wildcard filename |
| findnext | find next file with name matching wildcard filename |
| lseek | set position in file |
| open | open file |
| read | read from file |
| _read | read from file (no translation) |
| tell | find position in file |
| write | write to file |
| _write | write to file (no translation) |

These functions are not part of the draft ANSI standard, and it is usually preferable to use the `stdio` functions to provide portability to other machines. However, these functions are sufficiently standardized to be found in almost all C compilers.

Unbuffered files are referred to by means of an integer called the file handle. Handles 0, 1, 2, and 3 are reserved, and refer respectively to standard input, standard output, the auxiliary (serial) port, and the printer. Handles 4 and 5 are also reserved, so that handles allocated to user files start at 6. The Atari does not support a standard error stream, so the error handle is the same as the screen handle.

In order to use an unbuffered file, it is first opened or created using `open` or `creat`. It is preferable to use `open`, which has all the power of `creat` – `creat` should be viewed as obsolescent. These functions require information about the name of the file on disk, and the mode in which it is to be opened or created – this is specified using combinations of the macros defined in `fcntl.h`. If the file is successfully opened, a positive file handle, as described above, is returned. The C library also keeps a small amount of flags information about each file opened in this way, for example whether carriage returns should be removed on input (see the description of text mode versus binary mode in section 2.17). However, unlike the buffered files (usually called streams) defined in the header file `stdio.h` (see section 2.17), no information is buffered in memory by the C library.

Open unbuffered files should be closed automatically by the operating system when a program terminates. However, it is good practice to close them explicitly before terminating, or when they are no longer in use, especially as the number of available handles is limited.

Although the macro `fileno` is provided in `stdio.h` to give the handle associated with a buffered stream, it is in general unwise to use this handle for any of the unbuffered type operations defined in `io.h`, as these routines will not, for example, take into account any data already in the buffer before reading data from the disk.

## 2.9 Line A graphics functions <linea.h>

The `linea.h` header file defines the following "line A" graphics functions:

| | |
|---|---|
| A000 | initialize Line A |
| A001 | set point on screen |
| A002 | get colour of point on screen |
| A003 | draw line on screen |
| A004 | draw horizontal line |
| A005 | fill rectangle |
| A006 | fill polygon |
| A007 | bit block transfer |
| A008 | text block transfer |
| A009 | enable mouse cursor |
| A00A | disable mouse cursor |
| A00B | change mouse cursor form |
| A00C | clear sprite |
| A00D | enable sprite |
| A00E | copy raster form |

These functions provide access to the Atari's high speed graphics routines, called "Line A" because they make use of unimplemented 68000 instructions which start with the hexadecimal digit A. More information is available in Atari technical literature. For many purposes it is better to use the GEM VDI and AES bindings that are provided with Prospero C, which are more powerful and more portable, although not quite as fast.

## 2.10 Localization <locale.h>

The `locale.h` header file defines functions concerned with localization of the C run-time environment for a particular nationality. The function `setlocale` allows the current locale to be changed. However, Prospero C currently only supports the standard 'C' locale. Other locales, if supported, would affect, for example, which characters were considered to be letters in the `isalpha` function, and so on.

## 2.11 Mathematics <math.h>

The math.h header file defines the following mathematical functions:

| | |
|---|---|
| acos | computes arc cosine |
| asin | computes arc sine |
| atan | computes arc tangent |
| atan2 | computes arc tangent |
| ceil | find greatest integer below |
| cos | computes cosine |
| cosh | computes hyperbolic cosine |
| exp | computes exponentiation |
| fabs | computes absolute value |
| floor | finds least integer above |
| fmod | find remainder of division |
| frexp | separate exponent |
| ldexp | add exponent |
| log | computes natural logarithm |
| log10 | computes base 10 logarithm |
| modf | separate number into integral and fractional parts. |
| pow | computes power of |
| sin | computes sine |
| sinh | computes hyperbolic sine |
| sqrt | computes square root |
| tan | computes tangent |
| tanh | computes hyperbolic tangent |

and the macros EDOM, ERANGE and HUGE_VAL. In general, these functions take one or two parameters of type double, and return a double result. In the case of errors, they will set errno to either EDOM or ERANGE, as described in section 3.5.3

## 2.12 Process control <process.h>

The process.h header file defines the following functions:

| | |
|---|---|
| sleep | pause for a given time |
| spawn | execute program |
| spawnle | execute program with new environment |
| spawnlp | execute program with path search |
| spawnlpe | execute program with new environment and path search |
| spawnv | execute program |
| spawnve | execute program with new environment |
| spawnvp | execute program with path search |
| spawnvpe | execute program with new environment and path search |

The sleep function provides a delay for a given number of seconds, and is preferable to a loop as the delay period is portable. The spawn... functions all execute another program: the different functions make it easier to execute the program in the desired manner. They also allow passing of parameters.

## 2.13 Non-local jumps <setjmp.h>

The setjmp.h header file defines the following functions used for non-local jumps:

| | |
|---|---|
| setjmp | save execution environment |
| longjmp | jump to where execution environment was saved |

These functions enable an easy method of returning to a specific point in a program from anywhere else, usually when an error occurs. They provide a much easier escape route than returning an error indicator through several nested functions. See the description of the setjmp and longjmp functions for more information on how they are used.

## 2.14 Signal handling <signal.h>

The signal.h header file defines the following signal handling functions:

| | |
|---|---|
| signal | define how signals are to be handled |
| raise | force a signal to occur |

The signal function is used to specify what action to take when a signal occurs, while the raise function is used to cause a specific signal to occur. The header file also defines some macros suitable for use as parameters to one or other of these functions:

| | |
|---|---|
| SIG_DFL | default signal handling |
| SIG_ERR | error in call to signal |
| SIG_IGN | ignore signal |

| | |
|---|---|
| SIGABRT | abnormal termination |
| SIGFPE | erroneous arithmetic operation |
| SIGILL | invalid function image |
| SIGINT | receipt of an interactive attention signal |
| SIGSEGV | invalid memory access |
| SIGTERM | receipt of termination request |

## 2.15 Variable argument lists <stdarg.h>

The stdarg.h header file defines the following macros, needed for writing functions that have a variable number of parameters:

| | |
|---|---|
| va_start | called before using variable arguments |
| va_arg | get next variable argument |
| va_end | called after using all variable arguments |

These functions are used by functions that require a variable number of parameters such as printf, and allow the variable parameters to be read. It is up to the function to determine the type and number of parameters that are being passed; usually this is done by passing a count or format string in an earlier parameter, or by making the last parameter zero.

In the following example the function concat will concatenate an arbitrary number of strings together, placing them in the string given by the first parameter. The list is terminated by the NULL pointer.

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

char *concat (char *dest, ...)
{ va_list args;
  char *next;
  va_start(args, dest);
  next = va_arg(args, char *);
  while (next != NULL)
  { strcat (dest, next);
    next = va_arg(args, char *);
  }
  va_end(args);
  return dest;
}
```

## 2.16 Standard definitions <stddef.h>

The stddef.h header file defines the following commonly used types and macros:

| | |
|---|---|
| long ptrdiff_t | type of difference of two pointers |
| unsigned long int size_t | type of sizeof |
| NULL | pointer to no object |
| offsetof | offset of identifier within struct |
| errno | see section 2.6 |

This include file can be used when any of the standard types or macros above are used – note however that some of them are also defined in other header files. The offsetof macro can be used to find the offset of a variable within a structure.

## 2.17 Stream handling <stdio.h>

The stdio.h header file defines the following standard file handling macros and functions:

Macros:

| | |
|---|---|
| EOF | returned by many functions to indicate failure. |
| stdin | standard input stream (default keyboard) |
| stdout | standard output stream (default screen) |
| stderr | standard error stream (default screen) |
| stdaux | standard auxiliary stream |
| stdprn | standard printer stream |
| SEEK_SET | seek mode relative to the beginning of the file |
| SEEK_CUR | seek mode relative to the current position |
| SEEK_END | seek mode relative to the end of the file |
| _IOFBF | full buffered mode |
| _IOLBF | line buffered mode |
| _IONBF | unbuffered mode |

Functions:

| | |
|---|---|
| clearerr | clear stream error |
| fcloseall | close all the streams |
| feof | test if stream at end of file |
| ferror | test if stream error indicator set |
| fclose | close a stream |
| fflush | flush a stream |

| | |
|---|---|
| `fgetc` | get char from stream |
| `fgetpos` | get file position |
| `fgets` | get string from stream |
| `fileno` | find handle of a stream |
| `flushall` | flush all the streams |
| `fopen` | open a stream |
| `fprintf` | print to a file |
| `fputc` | put char to stream |
| `fputs` | put string to stream |
| `fread` | read array from stream |
| `freopen` | reopen a stream |
| `fscanf` | input from a file |
| `fseek` | set file position |
| `fsetpos` | set file position |
| `ftell` | get file position |
| `fwrite` | write array to stream |
| `getc` | get character from stream |
| `getchar` | get character from keyboard |
| `gets` | get string from screen |
| `perror` | print standard error |
| `printf` | print to the screen |
| `putc` | put char to stream |
| `putchar` | put char to screen |
| `puts` | put string to screen |
| `remove` | remove file |
| `rename` | rename filename |
| `rewind` | move to start of file |
| `scanf` | input from the keyboard |
| `setbuf` | set up buffer for a stream |
| `setvbuf` | set buffering mode |
| `sprintf` | print to a string |
| `sscanf` | input from a string |
| `tmpfile` | create temporary file |
| `tmpnam` | get temporary filename |
| `ungetc` | unget char from stream |
| `vfprintf` | print to a file from an argument list |
| `vprintf` | print to the screen from an argument list |
| `vsprintf` | print to a string from an argument list |

Standard file handling is based on the idea of a stream, or buffered file. A stream is referenced by an object of type `FILE` (also declared in `stdio.h`), although the user should only ever declare and use pointers to such objects. A `FILE` object contains all the information required by the library regarding a disk file with which the program is interacting, including information about the buffering, any buffered data, the GEMDOS file handle, and various flags.

Stream handling functions are sometimes referred to as buffered file handling operations in this manual, to distinguish them from the unbuffered file handling operations defined in `io.h`. However, it is possible to specify whether the files are to be buffered using the `setvbuf` and `setbuf` functions.

An important concept which affects the use of both streams and unbuffered files is that of text mode versus binary mode. Traditionally, C programs (particularly in the UNIX environment where C originated) assumed that lines of text in files were terminated by a single new-line character, whereas some operating systems, particularly on micros, place a carriage-return before the line-feed which terminates the line. In order to allow C programs to work under either system, the concept of text mode was invented. In text mode, lines of text are translated on input from the operating system's normal format (i.e., terminated by carriage-return/line-feed on GEMDOS) to the C standard format (i.e., terminated by line-feed only), and translated the other way on output.

When reading in binary data from a file, such considerations do not apply – indeed it would be unfortunate if all bytes whose value was 13 (the ASCII value of carriage-return) were skipped when reading a series of integers. Thus files which do not contain lines of text must be opened in binary mode.

Five `FILE` objects are predefined in Prospero C to refer to the standard input, output, error, auxiliary and printer handles, and their addresses are defined as the macros `stdin` to `stdprn`, listed above. Note that on the Atari there is no standard error handle, so that in fact the macro `stderr` also refers to `stdout`.

To use any file other than the predefined ones, a pointer to a `FILE` object is obtained using `fopen`. Information can then be read from or written to the file as appropriate. It is good practice to close all files when they are no longer in use, although Prospero C will flush and close all open streams at normal program termination.

## 2.18 Standard utilities <stdlib.h>

The stdlib.h include file defines a range of general utility functions, as follows:

| | |
|---|---|
| abort | abort the current process |
| abs | absolute value |
| atexit | exit trap |
| atof | convert string to float |
| atoi | convert string to int |
| atol | convert string to long |
| bsearch | binary search |
| calloc | allocate and clear memory |
| div | find quotient and remainder of division |
| ecvt | convert floating point to string |
| exit | terminate with clean up |
| _exit | terminate quickly |
| fcvt | convert floating point to string |
| free | free allocated memory |
| getenv | get environment variable |
| itoa | convert int to string |
| labs | find absolute value of long int |
| ldiv | quotient and remainder of long int division |
| ltoa | convert long int to string |
| malloc | allocate memory |
| qsort | sort a data array |
| rand | generate a random number |
| realloc | reallocate memory |
| srand | set seed for rand function |
| strtod | convert string to double |
| strtol | convert string to long int |
| strtoul | convert string to unsigned long int |
| swab | swap bytes |
| system | call system command processor |
| ultoa | convert unsigned long int to string |

These functions cover a wide range of assorted uses, including program termination, memory allocation, array sorting and searching, integer mathematical functions and functions to convert arithmetic types to and from strings. See the appropriate part of section 3 for details

## 2.19 String handling <string.h>

The string.h header file defines the following string and memory block manipulation functions:

| | |
|---|---|
| memccpy | copies memory up to a given character |
| memchr | find a character in a memory block |
| memcmp | compare two memory blocks |
| memcpy | copy a memory block |
| memicmp | compare two blocks, disregarding case |
| memmove | copy a memory block intelligently |
| memset | set a memory block to a value |
| strcat | concatenate two strings |
| strchr | find character in string |
| strcmp | compare two strings |
| strcoll | transform string for collating |
| strcpy | copy one string to another |
| strcspn | measure span of characters not in set |
| strdup | duplicate a string |
| strerror | map error number to string |
| stricmp | compare strings, case insensitive |
| strlen | measure length of string |
| strlwr | convert string to lower case |
| strncat | concatenate two strings, maximum length |
| strncmp | compare strings, length limited |
| strncpy | copy one string to another, length limited |
| strnicmp | compare two strings, ignoring case, max length |
| strnset | set string to value, max length |
| strpbrk | find break character in string |
| strrchr | find character not in string |
| strrev | reverse string |
| strset | set string to value |
| strspn | measure span of chars in set |
| strstr | locate occurrence of sub-string |
| strtok | get a token |
| strupr | convert string to upper case |

Functions whose names start `str...` deal with null-terminated strings; those starting `strn...` deal with null-terminated strings with a maximum length, and those starting `mem` deal with blocks of memory, with a size specified rather than a delimiting character. These functions provide most of the string handling functions that will normally be required, including concatenation and several routines that search the string for occurrence or non-occurrence of one or several characters. Those starting `mem...` are memory functions performing basic move and search functions on areas of memory.

It is up to the program to ensure that any pointers passed to receive results point to sufficiently large strings – no error checking is possible, and errors may result in program crashes.

If you have difficulty in remembering the order of the parameters in a function such as `strcpy`, remember that the first parameter is always the destination, as in an assignment expression `s1 = s2`.

e.g.,

```
#include <string.h>

main()

{   char buffer[81];  /* enough space for chars */

    puts("Enter a string");
    scanf("%80s", buffer);
    printf("forwards...\n%s\n", buffer);
    strrev(buffer);
    printf("backwards...\n%s\n", buffer);
    strrev(buffer);  /* back to normal */
}
```

## 2.20 Date and time <time.h>

The time.h header file defines the following date and time manipulation functions:

| | |
|---|---|
| asctime | convert time to string |
| clock | elapsed timer |
| ctime | convert time value to string |
| difftime | computes the difference between two times |
| gmtime | unpack Greenwich mean time |
| localtime | unpack local time |
| mktime | convert broken down time to calender time |
| strftime | general time string |
| time | get time |
| tzset | set time zone |

There are two formats for storing the time: the first is the internal GEMDOS format, corresponding to the type time_t; the second is a broken down structure, defined as follows:

```
typedef struct tm {int tm_sec;      seconds (0-59)
                   int tm_min;      minutes (0-59)
                   int tm_hour;     hours (0-23)
                   int tm_mday;     day of month (1-31)
                   int tm_mon;      month (0-11)
                   int tm_year;     year relative to 1900
                   int tm_wday;     day of week (Sun = 0)
                   int tm_yday;     day of year (0-365)
                   int tm_isdst;    daylight saving time flag
                  };
```

The strftime function can generate a string describing the date and time in a wide variety of formats.

# 3 LIBRARY FUNCTIONS AND MACROS

## 3.1 Introduction

The remainder of this volume details all the library functions defined in Prospero C, apart from the AES and VDI bindings, which are described in volumes III and IV.

## 3.2 Use of library functions

Normally when using the library functions, all the appropriate header files should be included at the start of the program. Failing to do so will not always result in compile-time errors, as in C it is not an error to use a function which was not previously declared. However, there are several disadvantages which may occur if a function is used without including the header file containing its prototype.

Firstly, any function which is called which was not previously declared will be assumed to return an `int`. If this is not in fact the case, the value returned will not be correct, and the program will behave unpredictably. If you specify the S option when compiling a program, such use of an undeclared function will generate a warning.

Secondly, if a function is called outside the scope of a function prototype, the compiler will not be able to check that the parameters you specify match in type and number the parameters expected by the function, and will only be able to guess at the conversions (if any) it should perform on the parameters before passing them. The default argument promotions will be performed, so that all `float` values are converted to `double`, and all `char` values to `int`. No library functions in fact expect parameters of type `char` or `float`, precisely because of the default conversions, but a function whose parameter should be `long int` would be passed an `int` if the argument was not explicitly stated to be `long` – this would lead to an unpredictable value being used. Particular care is needed with functions such as `malloc`, which traditionally has accepted an argument of type `int`, but in the draft ANSI standard expects an argument of type `size_t`, which is equivalent to `long`. A call such as `malloc(4)` will not work if `<stdlib.h>` is not included.

Thirdly, some functions are in fact defined as macros in the header files. Although the library also contains a true function corresponding to each such macro, which will be used if the header is not included, this is likely to result in larger and/or slower programs.

## 3.3 Character arguments

Many functions that might be expected to use `char` for parameters in fact use `int`. The reason for this lies in the default argument promotions – if a function had a parameter of type `char`, it could never be called in the absence of a function prototype, as all values of integer types narrower than `int` would be extended to type `int` before being passed. Declaring the parameter as being of type `int` in the prototype ensures that the argument is promoted in the same way whether or not the function prototype is present. If a `char` is passed to a function that is defined in this way, then it is converted to an `int` in the usual way, as described in volume I.

Note that the compiler option U (`char` is unsigned) does not affect the way in which library functions treat `char` values. For functions which operate on character values extended up to `int` before being passed, as described above, or return values of type `int` in the expectation that they will be truncated back to `char`, the way in which the extending and truncating is done will be determined by the setting of the U option in the program calling them. However, many of the string and memory comparison functions are not affected in this way, and always behave as if `char` was unsigned, as required by the C standard.

## 3.4 Functions and macros

Some functions in the library are defined as macros, rather than functions. If the header file is not included, then the macro will not be defined and the function will be called. It is also possible to undefine a macro using `#undef`. Again the function will be called. If a function is being assigned to a function pointer variable, or passed as a parameter, the function name will not be followed by a left parenthesis, and therefore the macro will not be invoked, and again it will be the true function that is referred to. For the same reason, if the function name in a function call is enclosed in brackets, this will ensure that a true function rather than a macro is invoked.

e.g. `toascii` can be defined as follows :-

```
int toascii(int c)
{  return c & 0x7f; }
```

or

```
#define toascii(c)  ((c) & 0x7f)
```

In the program segment :-

```
char a = toascii(193);

#include <ctype.h>

char b = toascii(193);
char c = (toascii)(193);

#undef toascii

char d = toascii(193);
```

the function will be called in the first, third and fourth use of `toascii`, and the macro definition will be used in the second. The individual function descriptions indicate when the header file defines a macro, and draw attention to the use of `#include` and `#undef`.

## 3.5 Handling errors

In some cases a function may be requested to do something that it cannot do. This may be because the parameters are unsuitable (e.g., `sqrt (-1)` ), or because of other factors (e.g., running out of memory or disk failures). When an error occurs the function needs to indicate that a error occurs. There are two standard ways of achieving this.

### 3.5.1 Function result

Some functions indicate an error by returning a value that could not result if the function was successful. Often this is –1, or `NULL` for functions which return pointers. For example, `open` returns –1 if it cannot open a file, and `malloc` returns `NULL` when there is not enough memory available.

### 3.5.2 The macro `errno`

The macro `errno` expands to a modifiable lvalue of type `int` (that is, it behaves as if it was declared as a variable of type `extern int`). Its value is initially zero when the program starts up, but many functions will set it to a positive error code upon detecting an error. No library function resets `errno` to zero, so that a program can call several functions, and then look at errno once at the end to see whether an error had occurred. The program would normally then reset errno to zero when it had corrected the error. The header file `<errno.h>` defines the macro `errno` and macros representing all possible values it may take from calls to the library functions. See the functions `strerror` and `perror` for ways of converting an error code into an error message.

When a function indicates an error by setting `errno`, it must still return a value. This value may by itself indicate that an error has occurred, as above, or it may be a value that cannot be distinguished from a valid function result except by examining `errno`.

### 3.5.3 Range and domain errors

The trigonometric and transcendental functions defined in the `<math.h>` header file can provoke three sorts of error. Firstly, the argument given can be a value where the mathematical function is not defined. This is known as a domain error, and will cause the value 0.0 to be returned, and `errno` to be set to the value of the macro `EDOM`.

e.g.,

```
asin(10.0)
```

Secondly, the argument can be such that the result, while mathematically defined, cannot be represented as a `double`, as its magnitude is too great. This is known as a range error, and will cause the value ±`HUGE_VAL` (depending on the sign of the true result) to be returned, and `errno` to be set to the value of the macro `ERANGE`.

e.g.,

```
exp(1e5)
```

The third class of errors occur for trigonometric functions, when the argument is so large that some or all significance is lost while reducing the argument to the range 0 to $2\pi$ prior to calculating the result. The C standard mandates that such errors should not be reported.

e.g.,

```
sin(1e5)
```

## 3.6 Library function descriptions

The functions are arranged in alphabetical order, with each function described in the format given below. Occasionally several functions are described together, especially where the functions can only sensibly be used together, or if they differ only very slightly in purpose. The layout for each function is as follows :-

# function name

A brief description of the function.

### Definition

Which header file needs to be included, and the prototype of the function as it appears in that header file.

### Purpose

A detailed description of the function and what the parameters (if any) mean.

### Returns

The value returned by the function, and how errors are indicated: usually via the value returned, or by setting `errno` appropriately (see above).

### Related Functions

A list of other functions that either perform a similar task, or are likely to be used in association with this function.

### Example

```
A simple example illustrating how the function is used.
These examples are not necessarily complete programs -
for instance few of them define main.
```

# a000..a00e

The a00*n* functions call the Atari "Line A" high speed graphics routines. They are not part of the draft ANSI standard.

## Definition

There are 14 "Line A" functions. In the definition *n* may be any of 0-9, A-E.

```
#include <linea.h>
long int a00n (struct RegRec *registers);
```

## Purpose

The functions a000 to a00e cause entry to the Atari's "Line A" graphics routines. These are fast graphic primitives which are executed using some of the MC68000 processor's unimplemented op-codes (the op-codes used all start with the hex digit 'A', hence the name). The interface to these routines was designed with the assembler programmer in mind, and therefore arguments are passed in registers rather than on the stack; the interface is also rather complicated, and is not particularly consistent between different routines. The procedures available are as described below; they are described in further detail in technical information available from Atari, or in various reference books on the Atari ST.

| Procedure | Purpose |
|---|---|
| a000 | Initialize Line A |
| a001 | Set point on screen |
| a002 | Get colour of point on screen |
| a003 | Draw line on screen |
| a004 | Draw horizontal line |
| a005 | Fill rectangle |
| a006 | Fill polygon |
| a007 | Bit block transfer |
| a008 | Text block transfer |
| a009 | Enable mouse cursor |
| a00a | Disable mouse cursor |
| a00b | Change mouse cursor form |
| a00c | Clear sprite |
| a00d | Enable sprite |
| a00e | Copy raster form |

The data for the machine registers is passed in `registers` which has type:-

```
struct RegRec { long int d0,d1,d2,d3,d4,d5,d6,d7;
                void *a0,*a1,*a2,*a3,*a4,*a5,*a6;  };
```

and refers to the 68000 registers. These values are copied into the machine registers before the "Line A" op-code is issued and copied back again afterwards.

### Returns

The `a00`*n* functions return the register `d0` after calling the "Line A" routine.

### Related functions

`gemdos, xbios, bios, VDI functions`

### Example

```
#include <linea.h>

struct RegRec registers;

/* get a pointer to the linea variables */
void *lineaptr = a000 (&registers);
```

# abort

## Definition

```
#include <stdlib.h>
void abort (void);
```

## Purpose

This function causes abnormal program termination. Functions registered with `atexit` are not called, open streams are not flushed or closed, and temporary files created with the `tmpfile` function are not removed. The signal `SIGABRT` is raised by calling the function `raise(SIGABRT)`, and if this returns, the program terminates with status `EXIT_FAILURE`.

## Returns

The `abort` function never returns to its caller.

## Related functions

`exit, _exit, raise`

## Example

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

main()
{  FILE *data = fopen("data.dat", "r");
   if (data == NULL)
   {  fprintf (stderr, "Can't open data file: %s \n",
               strerror(errno));
      abort();
   ...
}
```

# abs

## Definition

```
#include <stdlib.h>
int abs (int j);
```

## Purpose

This function returns the absolute value of the integer `j`.

## Returns

The `abs` function returns the absolute value of `j`. Note that the value of `abs(-32768)` cannot be expressed as a two-byte integer, and `abs` will return `-32768`. The value of `errno` will not be affected in this case, however.

## Related functions

`labs, fabs`

## Example

```
#include <stdlib.h>

main()
{   int i;

    for (i = -5; i < 5; i++)
        printf("%d\n", abs(i));
}
```

produces as output

```
5
4
3
2
1
0
1
2
3
4
```

# access

The access function checks whether a given file can be accessed. It is not part of the draft ANSI standard.

## Definition

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

## Purpose

This function determines whether the string pointed to by path is the name of a valid file or directory which can be accessed with the given mode.

| Mode | Access requirement |
|---|---|
| 0 | Check for existence only |
| 2 | Check for write access |
| 4 | Check for read access |
| 6 | Check for read and write access |

Under GEMDOS, all files have read permission, so modes 2 and 6 are equivalent, as are modes 0 and 4.

## Returns

The access function returns zero if the file or directory can be accessed with the given mode. Otherwise, –1 is returned, and errno will be set to either ENOENT, if the file does not exist, or EACCES, if it does not have the specified access permission.

## Related functions

chmod, open

## Example

```
#include <io.h>

if (access("DATA.IN", 0) )
{  puts("Can't find DATA.IN - terminating ");
   exit(1);
}
```

# acos

The acos function returns the arc cosine of a value.

## Definition

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

## Purpose

This function returns the principal value of the arc cosine of x. If x is not in the range -1 <= x <= 1, a domain error occurs. In this case, errno will be set to EDOM, and the value 0.0 will be returned.

## Returns

The acos function returns the arc cosine of x, expressed in radians, in the range 0 to $\pi$.

## Related functions

asin, atan, atan2

## Example

```
#include <math.h>

double x = 0.0;

printf ("pi / 2 = %f\n", acos (x) );
```

# asctime

The `asctime` function returns a string representing the given date and time.

## Definition

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

## Purpose

This function returns a pointer to a string describing the time represented by the structure pointed to by `timeptr`. This string always consists of 26 characters, including terminating new-line and null characters, laid out as in the following example :-

```
Wed Apr 14 11:04:22 1965\n\0
```

All fields have constant width.

## Returns

The `asctime` function returns a pointer to the string. The space occupied by the string will be overwritten by the next call to `asctime` or `ctime`.

## Related functions

`strftime, mktime, time, ctime`

## Example

```
#include <time.h>

main()
{   time_t now = time(NULL);

    printf("It is now %s\n", asctime(localtime(&now)));
}
```

# asin

The asin function returns the arc sine of a value.

## Definition

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

## Purpose

This function returns the principal value of the arc sine of x. If x is not in the range -1 <= x <= 1, a domain error occurs. In this case, errno will be set to EDOM, and the value 0.0 will be returned.

## Returns

The asin function returns the arc sine of x, expressed in radians, in the range $-\pi/2$ to $\pi/2$.

## Related functions

acos, atan, atan2

## Example

```
#include <math.h>

double x = 0.5;

printf (" pi / 6 = %f\n", asin (x) );
```

# assert

The `assert` macro is used for program verification.

## Definition

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

## Purpose

The `assert` macro is used in a program under development to verify that a particular condition which should be true at a particular point in the code is in fact true. If the value of `expression` is zero, a diagnostic message is output to the `stderr` stream, and execution is terminated by calling the `abort` function.

The diagnostic message is of the form

`Assertion` <exp> `failed: file` <filename>`, line` <lineno>

where <exp> is the expression which failed, and <filename> and <lineno> are the values of the standard `__FILE__` and `__LINE__` macros respectively.

Although the checks made by the `assert` macro are extremely useful when a program is under development or being debugged, they would add a significant amount to the size of a final program. In order to suppress the generation of these checks, the macro `NDEBUG` can be defined before including `assert.h`. In this case, the `assert` macro is defined as `(void) 0`, so that no code will be generated for any invocation of `assert`.

## Returns

There is no return value.

## Example

```
#include <assert.h>

   /* nextchar should never be NULL here */
   assert (nextchar != NULL);

   while (*nextchar && *nextchar != 's')
      nextchar++;
```

# atan

The `atan` function returns the arc tangent of a value.

### Definition

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

### Purpose

This function returns the principal value of the arc tangent of `x`.

### Returns

The `atan` function returns the arc tangent of `x`, expressed in radians, in the range $-\pi/2$ to $\pi/2$.

### Related functions

`acos, asin, atan2`

### Example

```
#include <math.h>

double x = 1.0;

printf (" pi / 4 = %f\n", atan(x) );
```

# atan2

The atan2 function returns the arc tangent of the quotient of two values.

## Definition

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

## Purpose

This function returns the arc tangent of y/x. If both x and y are zero, a domain error occurs. In this case, errno will be set to EDOM, and the value 0.0 will be returned.

## Returns

The atan2 function returns the arc tangent of y/x, expressed in radians, in the range $-\pi$ to $\pi$. The signs of both arguments determine the quadrant of the return value.

## Related functions

acos, asin, atan

## Example

```
#include <math.h>

double x = -1.0, y = 1.0;

printf (" - pi / 4 = %f\n", atan2 (y, x) );
```

# atexit

The `atexit` function causes a specified function to be executed at program exit.

## Definition

```
#include <stdlib.h>
int atexit (void (*func)(void));
```

## Purpose

This function registers the function whose address is given by `func`, causing it to be executed without arguments when normal program termination occurs. Functions registered in this way are called in reverse order of their registration, after execution of the `exit` function, or after the `main` function returns. At this point, all objects created with automatic storage duration during program execution are no longer in scope, so should not be accessed. However, open streams have not yet been closed, and temporary files have not been removed.

If program execution is terminated abnormally, by executing the `abort` or `_exit` functions, any functions registered using `atexit` will not be executed.

No more than 32 functions can be registered.

## Returns

The `atexit` function returns zero if the registration succeeds. If too many functions have been registered, a non-zero result is returned.

## Related functions

`exit, _exit, abort`

**Example**

```
#include <stdlib.h>

void tidy_up(void)
{ /* Clean up various matters */
  ...
  printf("Done\n");
}

void sign_off(void)
{ /* Print termination message */
  ...
  printf("Session terminated\n");
}

main()
{
  atexit(tidy_up);
  atexit(sign_off);
}
```

results in the following output :-

```
Session terminated
Done
```

# atof

The atof function converts a string to a double.

## Definition

```
#include <stdlib.h>
double atof(const char *nptr);
```

## Purpose

This function converts the initial portion of the string pointed to by nptr, which contains the decimal representation of an optionally signed floating point value to the corresponding double value.

This function has been superseded by strtod, but is provided for compatibility with older code. It is equivalent to the function call

```
strtod(nptr, NULL);
```

## Returns

The atof function returns the converted value, as a double. If the value represented by the string would cause overflow, the result is undefined. If the initial portion of the string (after skipping white space) does not correspond to a valid representation of a floating point value, zero is returned.

## Related functions

atol, strtol, sscanf

## Example

```
#include <stdlib.h>

double x;

x = atof ("3.141592653589793");

printf(" pi = %f\n", x);
```

# atoi

The `atoi` function converts a string to an integer.

### Definition

```
#include <stdlib.h>
int atoi(const char *nptr);
```

### Purpose

This function converts the initial portion of the string pointed to by `nptr`, which contains the decimal representation of an optionally signed integer to the corresponding integer value.

This function has been superseded by `strtol`, but is provided for compatibility with older code. It is equivalent (except on erroneous behavior) to the function call

```
(int) strtol(nptr, NULL, 10);
```

### Returns

The `atoi` function returns the converted value, as an `int`. If the value represented by the string is outside the range which can be represented as an `int`, the result will be undefined. If the first non-whitespace character in the string is not a digit or a sign character followed by a digit, zero is returned.

### Related functions

`atol, strtol, sscanf`

### Example

```
#include <stdlib.h>

int i;
char *string = "12345"

i = atoi (string);

printf ("String has value %d\n", i);
```

# atol

The `atol` function converts a string to a long integer.

### Definition

```
#include <stdlib.h>
long int atol(const char *nptr);
```

### Purpose

This function converts the initial portion of the string pointed to by `nptr`, which contains the decimal representation of an optionally signed integer to the corresponding long integer value.

This function has been superseded by `strtol`, but is provided for compatibility with older code. It is equivalent to the function call

```
strtol(nptr, NULL, 10);
```

### Returns

The `atol` function returns the converted value, as a `long int`. If the value represented by the string is outside the range which can be represented as a long integer, `errno` will be set to `ERANGE`, and an undefined result will be returned. If the first non-whitespace character in the string is not a digit or a sign character followed by a digit, zero is returned.

### Related functions

`atoi, strtol, sscanf`

### Example

```
#include <stdlib.h>

long int l;

l = atol ("1234567890");
```

# bios

The `bios` function calls an Atari BIOS function. It is not part of the draft ANSI standard.

## Definition

```
#include <dos.h>
long int bios (int funcno, ...);
```

## Purpose

This function is used to make a call to one of the Atari's BIOS functions. The `funcno` parameter specifies which BIOS function is required. Other parameters are passed where appropriate - their number, type, and purpose depend on the BIOS function requested. See Atari technical information for further details.

## Returns

The `bios` function returns the value returned in `d0` by the corresponding BIOS function.

## Related functions

`gemdos, xbios`

## Example

```
#include <dos.h>

/* Function to return the next key to be pressed */
int getch()

{    return ( bios (2,2) & 0xff)
}
```

# bsearch

The `bsearch` function searches a sorted array for a particular item.

## Definition

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void*, const void*));
```

## Purpose

This function searches the array whose base address is `base`, and which contains `nmemb` objects, each of size `size` bytes, for an object which compares equal to the object pointed to by `key`, using the comparison function `compar`.

The comparison function `compar` should return an integer less than, equal to or greater than zero, according to whether its first argument should be considered less than, equal to or greater than its second, and the objects in the array should be in ascending order as defined by this comparison function. The object is located using a binary search algorithm, which relies upon the objects being in order.

## Returns

The `bsearch` function returns a pointer to an object which matches `key`, if one can be found, otherwise `NULL`. If more than one object in the array would match `key`, any one of them may be returned.

## Related functions

`qsort`

**Example**

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

char *colors[] = {"amber", "green", "red" };
            /* Note the alphabetic order */

int mstrcmp(const void * a,const void * b)
{ return strcmp ( * (char **) a,* (char **) b);
}

main()

{ char ** color;
  char *key = "green";

  color = bsearch (&key, colors,
                   3, sizeof (char *), mstrcmp);

  printf ("We have found %s\n", color);
}
```

# calloc

The `calloc` function allocates and clears an area of memory.

## Definition

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

## Purpose

This function allocates and clears to zero enough memory for an array of `nmemb` objects, each of size `size` bytes. The memory can be released using `free` when it is no longer required. The memory will always start on an even address.

## Returns

The `calloc` function returns the address of the start of the allocated memory, or `NULL` if insufficient memory is available or zero bytes were requested.

## Related functions

`free, malloc, realloc`

## Example

```
#include <stdlib.h>
main()
{  int *a;

   /* Obtain space for 100 integers, and set them
      all to zero  */
   a = (int *) calloc(sizeof(int), 100);
   if (a == NULL)
      abort();
   ...
}
```

# ceil

The `ceil` function rounds a double value up to the nearest whole value.

### Definition

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

### Purpose

This function computes the smallest integral value which is not less than its argument `x`.

### Returns

The `ceil` function returns the `double` representation of the integer. There are no error cases.

### Related functions

`floor`

### Example

```
#include <math.h>

double truncate(double x) /* truncate towards zero */
{ if (x > 0.0)
      return floor(x);
  else
      return ceil(x);
}
```

# chdir

The `chdir` function sets the current working directory. It is not part of the draft ANSI standard.

## Definition

```
#include <direct.h>
int chdir (const char *pathname);
```

## Purpose

This function sets the default directory to be that specified by `pathname`. This may be either absolute or relative to the current default directory. If the pathname starts with a drive specifier (a drive letter followed by a colon) the current drive will also be changed. Subsequent references to file names will be interpreted relative to the new default directory.

Note that to include a backslash character in a string literal, two backslashes must be used, as the backslash character is used to introduce escape sequences. A forward slash may be used in place of the backslash character in `pathname` – these will be interpreted as if they were backslashes by the `chdir` function.

## Returns

The `chdir` function returns zero if the operation is successful. If the specified pathname does not exist, -1 is returned, and `errno` will be set to `ENOENT`. It is not an error to select a directory which is already current.

## Related functions

`mkdir, rmdir`

## Example

```
#include <direct.h>

chdir ("\\");  /* Make root directory current     */
chdir ("..");  /* Make parent directory current   */
chdir ("A:/"); /* Make root of drive A current    */
```

# chmod

The chmod function changes the access mode of a file. This function is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
int chmod (const char *path, int attribute);
```

## Purpose

The chmod function can be used to change the write protection of a file. Thus a file can be created, information written to it, and then protected. (This will only work with the soft write protect: if the write protect tab on the disk is open, then the disk cannot be written to).

The parameter attribute can be either S_IWRITE, to allow normal access (i.e. read and write), or S_IREAD, to make the file read only. S_IREAD and S_IWRITE are both defined in the header file fcntl.h.

## Returns

The chmod function returns the new attribute, or a negative number if an error occurs; in this case errno is set.

## Related functions

access, findfirst

## Example

```
#include <io.h>

if (access(filename,2) == -1)
   if (errno == EACCES)  /* write protected */
   {   chmod(filename, S_IWRITE);  /* allow write   */
       ...
   }
```

# clearerr

The `clearerr` function clears a stream's error and end-of-file indicators.

## Definition

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

## Purpose

This function clears the end of file and error indicators of the specified file `stream`. These are also cleared when the file is opened (using `fopen` or `freopen`) or rewound (using `rewind`), but no other file operations clear the error flag.

This is defined as a macro in `stdio.h`, but will be called as a function if `stdio.h` is not `#included`, or `clearerr` is undefined using `#undef`.

## Returns

The `clearerr` function returns no value.

## Related functions

`feof, ferror, perror`

## Example

```
#include <stdio.h>

   FILE * stream;

   if (ferror(stream))
   {  /* Correct the error */
      ...
      clearerr(stream);
   }
```

# clock

The `clock` function returns how long the program has been executing.

### Definition

```
#include <time.h>
clock_t clock(void);
```

### Purpose

This function returns the number of clock ticks which have occurred since the start of program execution. The number of clock ticks per second is defined by the macro `CLK_TCK`.

### Returns

The number of clock ticks is returned. For maximum portability, a program should check whether the value `(clock_t) -1` has been returned, indicating that this information is not available, although this will never be the case in Prospero C.

### Related functions

`time, difftime`

### Example

```
#include <time.h>

main()
{   int i, test_t, empty_t;
    clock_t start, end;

    start = clock();
    for (i = 0; i < 1000; i++)
       test();
    end = clock();
    test_t = (end - start)/CLK_TCK;
    start = clock();
    for (i = 0; i < 1000; i++);
    end = clock();
    empty_t = (end - start)/CLK_TCK;
    printf("1000 calls of test() took %d seconds.\n",
           test_t - empty_t);
}
```

# close

The `close` function closes an unbuffered file. This function is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
int close(int handle);
```

## Purpose

This function closes the file specified by handle. It is used for unbuffered files, rather than for streams (i.e. files opened using `open` or `creat` rather than `fopen`, and referred to by a handle rather than a `FILE *` pointer). The value of handle should be that returned by the call of `open` or `creat` when the file was opened.

## Returns

The `close` function returns zero if the file is successfully closed, otherwise –1. In the case of error, `errno` will be set to indicate the error. Normally the error code will be `EBADF`, indicating an invalid file handle.

## Related functions

`open, creat, dup, dup2`

## Example

```
#include <io.h>

main()
{ int handle = open ("scrndump", O_WRONLY);
  if (handle != -1)
  {  /* Output a few things to the file */
     ...
     close (handle);
  }
}
```

# cos

The cos function computes the cosine of a value.

## Definition

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

## Purpose

This function computes the cosine of x (in radians). If the value of x is large, the result may lose some or all significance (returning zero). This will not cause errno to be set.

## Returns

The cos function returns the cosine, in the range −1.0 to +1.0.

## Related functions

sin, tan

## Example

```
#include <math.h>

printf (" cos(pi) = %f\n", cos(3.1415926) );
```

# cosh

The cosh function computes the hyperbolic cosine of a value.

## Definition

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

## Purpose

This function computes the hyperbolic cosine of x.

## Returns

The cosh function returns the hyperbolic cosine. If the value of x is too large for the result to be represented, a range error occurs. In this case, errno will be set to the value ERANGE, and the value HUGE_VAL will be returned.

## Related functions

sinh, tanh

## Example

```
#include <math.h>

printf ("cosh (1) = %f\n", cosh (1.0) );
```

# creat

The creat function creates a new unbuffered file. This function is not part of the draft ANSI standard.

## Definition

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

## Purpose

This function creates a new unbuffered file with name filename, and access permission specified by pmode. If the given file already exists, it is truncated to zero length, and the access permission remains unchanged.

The values for the access permission are obtained by combining (using the bitwise OR operator |) one ore both of the following constants (defined in the fcntl.h header file) :-

| | |
|---|---|
| S_IWRITE | File can be written |
| S_IREAD | File can be read |

Under GEMDOS, write permission always implies read permission, so that S_IWRITE and S_IWRITE | S_IREAD are equivalent.

Files opened using creat are always in binary mode. The open function can be used to perform a similar task, but with the option of opening a file in text (translated) mode, by specifying the O_CREAT and O_TRUNC flags in the access parameter. It is recommended that open is used for new code – the creat function is provided for compatibility with existing code only.

## Returns

The creat function returns the handle of the file, which is used to refer to the file in subsequent read, write, seek, dup, dup2 or close function calls. If an error occurs, –1 is returned, and errno will be set.

## Related functions

open, close, read, write, seek, dup, dup2

**Example**

```
#include <io.h>

main()
{ int handle = creat ("temp.$$$", S_IWRITE | S_IREAD);

  /* use the temporary file */

  close (handle);

  remove ("temp.$$$");

}
```

# ctime

The ctime function converts a date and time to a character string.

### Definition

```
#include <time.h>
char *ctime(const time_t *timer);
```

### Purpose

This function converts the calendar time pointed to by timer (treated as local time) into a string. It is equivalent to

```
asctime (localtime (timer))
```

### Returns

The ctime function returns the address of the string. This always contains 26 characters, in the same format as returned by the asctime function. The buffer containing the string will be overwritten by the next call of asctime or ctime.

### Related functions

asctime, localtime, time

### Example

```
#include <time.h>

main()
{  time_t now = time(NULL);

   printf("It is now %s\n", ctime (&now));
}
```

# difftime

The `difftime` function calculates the difference between two calendar times.

## Definition

```
#include <time.h>
double difftime (time_t time1, time_t time2);
```

## Purpose

This function calculates the difference (in seconds) between the two calendar times `time2 - time1`. The type `time_t` (defined in `time.h`), is a compressed form containing the year, month, day, hour, minute and second, so that simply subtracting two `time_t` values would not give a meaningful result.

## Returns

The `difftime` function returns the elapsed number of seconds between `time1` and `time2`, expressed as a `double`. If the value is negative, `time2` was earlier than `time1`.

## Related functions

`time, clock`

## Example

```
#include <time.h>

main()
{  time_t start, end;

   time(&start);
   test();
   time(&end);

   /* Note: This could be achieved more efficiently
      using clock() */

   printf("test() took %g seconds.\n",
          difftime ( start, end) );
}
```

# div

The div function computes the quotient and remainder of an integer division.

## Definition

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

## Purpose

This function calculates the quotient and remainder of the division of numer by denom. The sign of the remainder is always the same as the sign of the quotient (unless the remainder is zero). If denom is zero, or numer is MIN_INT and denom is –1, the result is undefined.

## Returns

The div function returns a structure with two integer fields, quot and rem. The type div_t is defined in stdlib.h.

## Related functions

ldiv

## Example

```
#include <stdlib.h>

div_t result;
int hours, minutes;

result = div (minutes, 60);
hours = result.quot;
minutes = result.rem;
```

# drivemap

The `drivemap` function finds which drives are available. It is not part of the draft ANSI standard.

### Definition

```
#include <direct.h>
unsigned int drivemap(void)
```

### Purpose

This function finds which disk drives are connected to the computer (on a single disk Atari it will still find drives A and B as these both exist logically).

### Returns

This function returns a bitmap where a 1 indicates a drive is present and a 0 indicates it isn't. Drive A is bit 0, drive B is bit 1 and so on.

### Related functions

`getdfs, getdisk, setdisk`

### Example

```
#include <direct.h>

int i;
unsigned int map = drivemap();

for (i=0;i<16;i++)
   if (map & (1 << i))
       printf ("Drive %c is available\n", i + 'A');
```

# dup

The dup function duplicates a file handle for an unbuffered file. This function is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
int dup (int handle);
```

## Purpose

This function returns a new file handle which refers to the same file as specified by handle, which must be one of the standard handles. Subsequent operations on the file can use either handle.

## Returns

The dup function returns the new file handle. If an error occurs, –1 is returned, and errno is set to indicate the type of error.

## Related functions

dup2, fileno, open, creat

## Example

```
#include <io.h>

int screen_handle, printer_handle;
int handle, hard_copy;

if (hard_copy)
    handle = dup (printer_handle);
else
    handle = dup (screen_handle);
    /* output via handle will go to screen unless
       hardcopy is requested                          */
```

# dup2

The dup2 function duplicates a file handle for an unbuffered file. This function is not part of the draft ANSI standard.

### Definition

```
#include <io.h>
int dup2 (int handle, int new_handle);
```

### Purpose

This function forces the handle new_handle to refer to the same file as that specified by handle, which must be one of the standard handles. If new_handle already refers to an open file, it will be closed first. Subsequent operations on the file can use either handle.

### Returns

The dup2 function returns the new file handle. If an error occurs, –1 is returned, and errno is set to indicate the type of error which. Otherwise, the new file handle will be new_handle.

### Related functions

dup, fileno, open, creat

### Example

```
#include <io.h>

int screen_handle, printer_handle, printer;

if (printer)
    dup2 (printer_handle, screen_handle);

   /* If printer then direct screen output there */
```

# ecvt

The ecvt function converts a double value to a string of ASCII digits. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
char *ecvt (double value, int ndigits,
            int *decptr, int *signptr);
```

## Purpose

This function converts its argument value into ASCII digits in a static buffer, and returns a pointer to these digits. The first ndigits digits will be placed in the string, followed by a null character, and the value will be rounded to this number of digits (c.f. fcvt, where ndigits refers to the number of digits after the decimal point). If value is negative, a non-zero integer is stored at the location pointed to by signptr, otherwise zero is stored. The number of significant digits of value which precede the decimal point is returned in the integer pointed to by decptr. Note that neither the sign nor the decimal point are stored in the string, which contains only digits.

Note that it is usually better to use printf which is easier to use, and being part of the ANSI standard, is more portable. However, ecvt may be preferable in low-level applications, such as an alternative printf command.

## Returns

The ecvt function returns a pointer to the static string containing the characters. The contents of this string will be overwritten by the next call of ecvt or fcvt, including calls by other library functions such as printf. Note that the string only has space for 18 characters, and if ndigits is greater than 18, it will be treated as if it was 18. A double value will never have more than 18 significant digits.

## Related functions

fcvt, printf

**Example**

```
#include <stdlib.h>

int expt, sign;
double value = -5.0;

char *buffer = ecvt (value, 6, &expt, &sign);

if (sign != 0)
    putchar ('-');
printf ("%c.%se%+03d\n", *buffer, buffer + 1,expt - 1);
```

Produces the output

```
-5.00000e+00
```

# eof

The `eof` function tests for end of file on an unbuffered file. It is not part of the draft ANSI standard.

### Definition

```
#include <io.h>
int eof (int handle);
```

### Purpose

This function determines whether the position of the file associated with handle is at the end of the file. This would typically be used to test whether more input was available before attempting to read data from a file.

### Returns

The `eof` function returns 1 if the file is at the end, zero if it is not. If `handle` is not a valid file handle of a currently open file, –1 is returned, and `errno` is set to `EBADF`.

### Related functions

`feof`

### Example

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

char buffer[512];

int handle = open ( filename, O_RDONLY, 0);
   while (! eof (handle))
      read (handle, buffer, 512);
```

# exit

The `exit` function terminates execution of a program.

### Definition

```
#include <stdlib.h>
void exit (int status);
```

### Purpose

This function terminates execution of a program, in the same way as if the `main` function had returned (normal termination). Any functions registered using the `atexit` function will be executed (in reverse order of registration). After this, all open streams are flushed then closed, and all temporary files created with the `tmpfile` function are removed. The program then terminates, returning the value of status to the program which caused it to be executed. Under GEMDOS, only the low-order byte of this value is returned.

Normally a status of zero would be used to indicate success, and higher values to indicate increasingly severe error conditions. The macros `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `stdlib.h`.

### Returns

The `exit` function never returns to its caller.

### Related functions

`atexit, abort, _exit`

### Example

The following returns success or failure to the calling program.

```
#include <stdlib.h>

int error = 0;

   if (error)
        exit (EXIT_FAILURE);
   else
        exit (EXIT_SUCCESS);
```

# _exit

The `_exit` function terminates execution of a program. This function is not part of the draft ANSI standard.

### Definition

```
#include <stdlib.h>
void _exit (int status);
```

### Purpose

This function terminates execution of a program, but does not perform the normal closing down procedures. Functions registered using `atexit` are not executed, open streams are not flushed or closed, and temporary files are not removed. The value of `status` is returned the program which caused it to be executed. Under GEMDOS, only the low-order byte of this value is returned.

Normally a status of zero would be used to indicate success, and higher values to indicate increasingly severe error conditions. However, `_exit` would normally only be used for severe error conditions where it was not worth trying to exit cleanly.

### Returns

The `_exit` function never returns to its caller.

### Related functions

`abort, exit`

### Example

If a disaster happened, this program will terminate and indicate failure to the calling program.

```
#include <stdlib.h>

int disaster;

if (disaster)
   _exit (EXIT_FAILURE);
```

# exp

The exp function returns the exponential of a value.

## Definition

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

## Purpose

This function computes the exponential of x.

## Returns

The exp function returns the exponential. If x is too large for the result to be represented as a double, a range error occurs. In this case, errno will be set to ERANGE, and the value HUGE_VAL will be returned.

## Related functions

log

## Example

```
#include <math.h>

printf ("e = %f\n", exp (1.0) );
```

# fabs

The `fabs` function returns the absolute magnitude of a floating point value.

### Definition

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

### Purpose

This function computes the absolute value of the `double` value `x`.

### Returns

The `fabs` function returns the absolute value of `x`, as a `double`. There is no error return.

### Related functions

`abs, labs`

### Example

```
#include <math.h>

printf ("The absolute value of -1.234 is %f\n",
                                fabs (-1.234) );
```

produces the output

```
The absolute value of -1.234 is 1.234000
```

# fclose

The `fclose` function closes a stream.

### Definition

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

### Purpose

This function flushes then closes the stream `fp`. Any buffer allocated by the system (rather than assigned using `setvbuf` or `setbuf`) will be freed, and files created using `tmpfile` will be deleted. All streams other than the predefined standard streams are automatically closed at program termination.

### Returns

The `fclose` function returns zero if no errors were detected, otherwise it returns `EOF`, setting `errno` to indicate the nature of the error.

### Related functions

`fopen, freopen, tmpfile, fcloseall, fflush, flushall`

### Example

```
#include <stdio.h>

update_workfile()
{ FILE *workfile;

  workfile = fopen("workfile.dat","w");

  /* Now write the required data to workfile */

  /* Must close workfile before leaving function */
  fclose (workfile);
}
```

# fcloseall

The `fcloseall` function closes all streams opened by the program. This function is not part of the draft ANSI standard.

## Definition

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

## Purpose

This function flushes and closes all open streams other than the predefined standard streams `stdin`, `stdout`, `stderr`, `stdaux` and `stdprn`. Any automatically allocated buffers are released, and temporary files created by `tmpfile` are removed, exactly as if `fclose` had been called for each open stream. Note that all open streams are automatically closed on normal program termination.

## Returns

The `fcloseall` function returns the number of streams successfully closed. If an error is detected, `EOF` is returned, and `errno` is set to indicate the error.

## Related functions

`fopen, freopen, tmpfile, fclose, flushall`

## Example

```
#include <stdio.h>

main()

{ int quitting = 0;
  /* open and use plenty of files */

  if (quitting)
    if (fcoseall() == EOF)
      perror("Error closing files");
}
```

# fcvt

The `fcvt` function converts a double value to a string of ASCII digits. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
char *fcvt (double value, int ndigits,
            int *decptr,  int *signptr);
```

## Purpose

This function converts its argument `value` into ASCII digits in a static buffer, and returns a pointer to these digits. The parameter `ndigits` specifies how many digits after the decimal point are to be converted, and the value will be rounded to this number of decimal places (c.f. `ecvt`, where `ndigits` refers to the total number of digits to be printed). If `value` is negative, a non-zero integer is stored at the location pointed to by `signptr`, otherwise zero is stored. The number of significant digits of `value` which precede the decimal point is returned in the integer pointed to by `decptr`. Note that neither the sign nor the decimal point are stored in the string, which contains only digits.

It is usually better to use `printf` which is easier to use, and being part of the ANSI standard, is more portable. However, `fcvt` may be preferable in low-level applications, such as an alternative `printf` command.

## Returns

The `fcvt` function returns a pointer to the static string containing the characters. The contents of this string will be overwritten by the next call of `ecvt` or `fcvt`, including calls by other library functions such as `printf`. Note that the string only has space for 18 characters. A `double` value will never have more than 18 significant digits.

## Related functions

`ecvt, printf`

**Example**

```
#include <stdlib.h>

int expt, sign;
double value = -5.0;

char *buffer = fcvt (value, 4, &expt, &sign);

if (sign != 0)
    putchar ('-');
printf ("%c.%se%+03d\n", *buffer, buffer + 1,expt - 1);
```

Produces the output

```
-5.0000e+00
```

# feof

The `feof` function tests whether a stream has reached the end of a file.

## Definition

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

## Purpose

This function tests the end-of-file indicator for the specified file. This indicator is set when a read operation fails because there is no more data in a file, and cleared when the file position is moved using `rewind`, `fseek`, or `fsetpos`, or when the program explicitly clears it using `clearerr`.

Note that this is declared as a macro in `stdio.h`. If `stdio.h` is not included, or if `feof` is `#undef`'d, a library function will be called.

## Returns

The `feof` function returns zero if the end-of-file indicator is not set, and non-zero if it is set.

## Related functions

`clearerr, ferror, eof, fread`

## Example

```
#include <stdio.h>

double process_data(FILE *stream)
{  double answer;
   if (feof (stream))
      return -1;   /* no more data to process */

   /* read some data and process it */
   ...
   return answer;
}
```

# ferror

The `ferror` function tests whether an error has been detected for a stream.

### Definition

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

### Purpose

This function tests the error indicator for the specified file. This indicator is set when a read/write error occurs, and cleared by calling `rewind` or `clearerr`.

Note that this is declared as a macro in `stdio.h`. If `stdio.h` is not included, or if `ferror` is `#undef`'d, a library function will be called.

### Returns

The `ferror` function returns zero if the error indicator is not set, and non-zero if it is set. The result is undefined if `stream` is not valid.

### Related functions

`clearerr, feof, rewind`

### Example

```
#include <stdio.h>

main()

{  FILE *stream = fopen ("myfile.dat", "w");

   fwrite (stream ,data, size, number);
   if (ferror (stream) )
      { rewind (stream);
        fwrite (stream ,data, size, number);
        /* Try once more */
      }
}
```

# fflush

The `fflush` function flushes a stream's buffer.

### Definition

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

### Purpose

If the file referred to by `stream` has been opened for output or update, the contents of its output buffer (if any) are written to the disk file or device. If a file has been opened for update, the next file operation after a `fflush` can be either input or output.

Note that buffers are flushed automatically when they are full, when the file position is moved using `fseek`, `fsetpos` or `rewind`, or when a stream is closed.

If `stream` is a `NULL` pointer then all open streams are flushed in the manner described above.

### Returns

The `fflush` function returns zero if the operation was successful. In the case of error, `EOF` is returned, and `errno` will be set to indicate the error.

### Related functions

`fseek, fsetpos, rewind, flushall, fclose`

### Example

```
#include <stdio.h>

main()

{   FILE * stream = fopen ("myfile", "w");

    /* Output some data */

    fflush (stream);
    /* make sure data is written to stream */
}
```

# fgetc

The `fgetc` function reads a character from a stream.

## Definition

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

## Purpose

This function gets the next character from the file referred to by `stream`.

## Returns

The `fgetc` function returns the next character (converted to an `int`). If the stream is at end-of-file, the stream's end-of-file indicator will be set. If a read error occurs, the stream's error indicator is set. In both these cases, `fgetc` will return `EOF`.

## Related functions

`feof, ferror, getc, getchar, fopen`

## Example

```
#include <stdio.h>

FILE * stream = fopen ("myfile", "r");

char ch = fgetc (stream);
```

# fgetpos

The `fgetpos` function stores the current file position of a stream.

## Definition

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

## Purpose

This function stores the current position of the file referred to by `stream` into the object pointed to by `pos`, in a form suitable for passing to `fsetpos` to restore the file position to that stored. The type `fpos_t` is defined in `stdio.h`.

## Returns

The `fgetpos` function returns zero if successful, otherwise a non-zero result will be returned, and `errno` will be set to indicate the error.

## Related functions

`fsetpos, ftell, fseek`

## Example

```
#include <stdio.h>

FILE * stream;

fpos_t pos;

fgetpos (stream, &pos);

rewind (stream);  /* Back to beginning of the file */

fsetpos (stream, &pos); /* and back to where we were */
```

# fgets

The `fgets` function reads a string of characters from a stream.

## Definition

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

## Purpose

This function reads characters from the file referred to by `stream` into the string pointed to by `s`, until `n - 1` characters have been read, or the last character read into `s` was a newline character, or the end of the file is reached. A null character is then always appended to the string `s`.

## Returns

The `fgets` function returns the pointer `s` if successful. If end-of-file is encountered before any characters have been read into `s`, `NULL` is returned and `s` is not modified. If a read error occurs, `NULL` is returned but the contents of `s` will have been modified.

## Related functions

`fgetc, fputs, gets, puts`

## Example

```
#include <stdio.h>

void print_text_file (const char * file)

{  FILE *stream = fopen (file, "r");
   char buffer[256];

   if (stream != NULL)
       while (fgets (buffer, 256, stream) != NULL)
            puts (buffer);
   fclose (stream);
}
```

# filelength

The `filelength` function calculates the size of an unbuffered file. It is not part of the draft ANSI standard.

### Definition

```
#include <io.h>
long int filelength (int handle);
```

### Purpose

This function computes the length of an unbuffered file associated with the given `handle`.

### Returns

The `filelength` function returns the length of the file in bytes. If an error occurs, `-1L` is returned, and `errno` will be set to indicate the error which has occurred.

### Related functions

`access, fileno, getdfs`

### Example

```
#include <io.h>

int edit ( int handle)

{  if ( filelength ( handle) ) > 10000)
       return (-1);     /* file too long */

   /* edit the file */
}
```

# fileno

The `fileno` function returns the file handle associated with a stream. It is not part of the draft ANSI standard.

### Definition

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

### Purpose

This function obtains the operating system handle associated with the file referred to by `stream` when it was first opened. This might be required for using lower level (unbuffered) i/o functions or operating system calls with a file originally opened as a stream (buffered). Such mixing of modes for a file should be done with care.

Note that `fileno` is defined as a macro in `stdio.h`, but will be called as a true function if `stdio.h` is not included, or if `fileno` is `#undef`'d.

### Returns

The `fileno` function returns the file handle. If `stream` does not refer to a valid open file, the result is undefined.

### Related functions

`fopen, freopen, dup, dup2`

### Example

```
#include <stdio.h>

long int length (FILE *stream)

{   int handle = fileno (stream);
    return (filelength (handle) );
}
```

# findfirst, findnext

The `findfirst` and `findnext` functions find all files in the current directory that satisfy a filename with wildcards. They are not part of the draft ANSI standard.

## Definition

```
#include <io.h>
int findfirst (const char *pathname, struct dta *info,
               int attribute);
int findnext (struct dta *info);
```

## Purpose

These functions provide a means of finding all files in the current directory that match the given `pathname`, which may include the wildcard characters '`?`' and '`*`'. The '`?`' means any letter, and the '`*`' means any group of letters. Thus `ABC?.C` will include `ABC1.C` but not `ABC10.C`; wheras both of these will be included using `ABC*.C`. The most general pathname is `*.*` which will include all files in that directory. The `attribute` defines which attributes the files must have to be included.

The value of `attribute` can be used to find files with read and write access, with read access only, or for finding sub-directories. The macros `S_IREAD` and `S_IWRITE` (in `fcntl.h`) define read-only and read or write files respectively. These can be combined (using `|`) with `S_SUBDIR` to find subdirectories.

To find the files, on the first occasion use `findfirst` and subsequently use `findnext`. Only `findfirst` requires the `pathname` and `attribute` as these are the same for subsequent calls of `findnext`.

The file found is given by `info` which is a pointer to the structure `dta`, which is defined:-

```
struct dta {
           char attrib;      /* attribute found */
           long time;        /* date and time of file */
           long length;      /* length of file */
           char filename[13];
           char extra[22];   /* used by OS */
           };
```

### Returns

Both functions return zero if a file or another file has been found and non-zero if no file is found. A disk error, or no file being found with `findfirst` will cause `errno` to be set (the error being `ENOENT` if file is not found with `findfirst`).

### Related functions

`access, chmod`

### Example

```
/* program to print out the filenames of all files */
/* in the current directory                        */

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

main()
{  struct dta info;

   if (findfirst("*.*", &info, S_IWRITE | S_IREAD)==0)
      do
        puts(info.filename);
      while (findnext( &info) == 0);
   else
      /* didn't find any */
      perror("Disk error");
}
```

# floor

The `floor` function returns the largest integer not greater than a value.

### Definition

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

### Purpose

This function computes the largest integral value which is not greater than the argument `x`.

### Returns

The `floor` function returns the integer, expressed as a `double`. There are no error cases.

### Related functions

`ceil`

### Example

```
#include <math.h>

double e = 2.7818;

printf ("floor(e) = %0.1f, but floor(-e) = %0.1f\n",
                              floor (e), floor (-e) );
```

will produce

```
floor(e) = 2.0, but floor(-e) = -3.0
```

# flushall

The `flushall` function flushes all open streams. It is not part of the draft ANSI standard.

### Definition

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

### Purpose

This function flushes all open streams (including the predefined streams `stdin`, `stdout`, `stderr`, `straux` and `stdprn`), in the same way as if `fflush` had been called for each open stream in turn. It is equivalent to

```
fflush (NULL);
```

except for the value returned. Any buffered output data is written to the corresponding file or device, and any buffered input data is discarded, to be re-read by the next input operation. The streams are not closed.

Note that buffers are flushed automatically when they are full, when the file position is moved using `fseek`, `fsetpos` or `rewind`, or when a stream is closed.

### Returns

The `flushall` function returns the number of open streams. If any errors occurred while flushing, `errno` will be set.

### Related functions

`fflush`, `fcloseall`

### Example

```
#include <stdio.h>

main()
{   /* open some files and read and write some data */
    flushall();  /* all data is now on disk */
    ...
}
```

# fmod

The `fmod` function calculates the remainder from a floating point division.

## Definition

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

## Purpose

This function computes the floating point remainder of `x/y`.

## Returns

The `fmod` function returns the remainder `f`, where `x = f + i*y` for some integer `i`. The result has the same sign as `x`, and magnitude less than `y`. If `x/y` cannot be represented (for example, `y` is zero), the result is undefined.

## Related functions

`ceil, fabs, floor, modf`

## Example

```
#include <math.h>

double price = 9995.0;    /* price in pence */

double pence = fmod (price, 100.0);
```

# fopen

The `fopen` function opens a stream for subsequent reading or writing.

## Definition

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

## Purpose

This function opens the named file `filename` as a stream (i.e. for buffered i/o), in a mode specified by the contents of the string pointed to by `mode` as follows :-

| Mode | Meaning |
|---|---|
| `"r"` | Open text file for reading. |
| `"w"` | Create new text file for writing, or truncate to zero length. |
| `"a"` | Open or create text file for writing at end of file only. |
| `"rb"` | Open binary file for reading. |
| `"wb"` | Create new binary file for writing, or truncate to zero length. |
| `"ab"` | Open or create binary file for writing at end of file only. |
| `"r+"` | Open text file for update. |
| `"w+"` | Create new text file for update, or truncate to zero length. |
| `"a+"` | Open or create text file for update, writing at end of file only. |
| `"r+b"` | Open binary file for update. |
| `"w+b"` | Create new binary file for update, or truncate to zero length. |
| `"a+b"` | Open or create binary file for update, writing at end of file only. |
| `"rb+"` | Open binary file for update. |
| `"wb+"` | Create new binary file for update, or truncate to zero length. |
| `"ab+"` | Open or create binary file for update, writing at end of file only. |

Any characters which occur after one of the above sequences will be ignored.

For read modes, the named file must exist, while append or write modes will create the file if it does not already exist. In write mode, if the file already exists, the previous contents are discarded.

In update modes, both reading and writing are permitted on the stream, but an output operation should not be followed by an input operation without an intervening call of `fflush` or a file positioning operation (`fseek`, `rewind` or `fsetpos`) on the stream, and an input operation should not be followed by an output operation unless `fflush` or a file positioning operation has been issued, or the input has encountered end-of-file.

For text files, newline characters (`'\n'`) are translated into carriage-return / newline sequences on output, and carriage-return characters are stripped on input. This translation does not occur for binary files.

### Returns

The `fopen` function returns the file pointer of the opened stream. If the file could not be opened, `NULL` is returned, and `errno` is set to indicate the error which occurred.

### Related functions

`fclose, fcloseall, freopen`

### Example

```
#include <stdio.h>

FILE *stream = fopen ("temp.$$$", "wb");

/* write data to temporary file */
```

# fprintf

The `fprintf` function writes formatted output to a stream.

### Definition

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

### Purpose

This function writes a string of characters controlled by the string pointed to by `format` to the file pointed to by `stream`. See the description of the function `printf` for details on the `format` string. The function takes a variable number of arguments – the type and number of the arguments after the `format` argument is determined by the layout of the string pointed to by `format`.

### Returns

The `fprintf` function returns the number of characters written to the stream. If a write error occurs, a negative value is returned, and `errno` will be set.

### Related functions

`printf, sprintf, vfprintf, vprintf, vsprintf`

### Example

```
#include <stdio.h>

FILE *stream = fopen ("data", "w");

int age = 21;
char *name = "Fred";

fprintf (stream, "%s is %d", name, age);
```

# fputc

The `fputc` function writes a character to a stream.

### Definition

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

### Purpose

This function writes the character `c` to the stream described by `stream`.

### Returns

The `fputc` function returns the character written. If a write error occurs, `EOF` is returned and the file error flag and `errno` will be set.

### Related functions

`putc, putchar, fputs`

### Example

```
#include <stdio.h>

FILE *stream = fopen ("file", "w");

fputc ('\n', stream);
```

# fputs

The `fputs` function writes a string of characters to a stream.

## Definition

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

## Purpose

This function writes a string of characters pointed to by `s` to the file described by `stream`, up to but not including the terminating null char.

## Returns

The `fputs` function returns zero if successful. If a write error occurs, a non-zero value is returned, and the stream's error indicator and `errno` will be set.

## Related functions

`fputc, putc, puts`

## Example

```
#include <stdio.h>

FILE *stream = fopen ("file", "w");

fputs ("Hello", stream);
```

# fread

The `fread` function reads data from a stream into an array.

## Definition

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

## Purpose

This function reads up to `nmemb` objects, each of `size` bytes, from the stream described by `stream` into the array whose base is given by `ptr`. If the stream was opened in text mode, carriage returns in the input will be removed. This is unlikely to be what was intended.

## Returns

The `fread` function returns the number of objects successfully read – this may be less than `nmemb` if a read error or end-of-file was encountered. If an error occurred, the stream's error indicator and `errno` will be set; if end-of-file is encountered, the stream's end-of-file indicator is set.

## Related functions

`fopen, fwrite, read, write`

## Example

```
#include <stdio.h>

FILE *stream = fopen ("data", "rb");

double data[100];

fread (data, sizeof (double), 100, stream);
```

# free

The `free` function releases storage allocated using `malloc`.

## Definition

```
#include <stdlib.h>
void free (void *ptr);
```

## Purpose

If `ptr` is not `NULL`, this function causes the storage it points to (which must have been previously allocated using `malloc`, `calloc` or `realloc`) to be released and made available for re-use. If `ptr` is `NULL`, the function does nothing.

## Returns

The `free` function returns no value. If `ptr` was not previously allocated using `malloc`, `calloc` or `realloc`, or has already been released, unpredictable results will occur.

## Related functions

`calloc, malloc, realloc`

## Example

```
#include <stdlib.h>

main()

{   char *buffer = malloc (16384);

    /* use the buffer */

    free (buffer);
}
```

# freopen

The `freopen` function associates a stream with a file or device, closing any file it was previously associated with.

### Definition

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

### Purpose

This function closes any file currently associated with `stream`, and then opens the file specified by `filename` in the mode specified by `mode`, and associates that file with `stream`. The permitted values of the `mode` parameter and their meanings are described under the `fopen` function.

This function is often used to make one of the standard streams `stdin`, `stdout`, `stderr`, `stdaux` or `stdprn` refer to a disk file rather than a device.

### Returns

The `freopen` function returns a pointer to the newly opened file if successful (this is the same as `stream`). If an error occurs, a `NULL` pointer is returned, and `errno` will be set. Any errors which occur while trying to close the file previously associated with `stream` are ignored.

### Related functions

`fopen, fclose`

### Example

```
#include <stdio.h>

int logfile = 0;

main()

{  if (hardcopy)
       freopen ( "myfile.log", "w", stdout);

   /* perform some output */
}
```

# frexp

The `frexp` function breaks a floating-point value into a normalized fraction and a binary exponent.

## Definition

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

## Purpose

This function breaks the floating-point value `value` into a normalized fraction and an integral power of two by which the fraction is multiplied. The power of two is stored in the object pointed to by `exp`.

## Returns

If value is zero, both the function result and the value stored in `*exp` are zero. Otherwise, the `frexp` function returns the normalized fraction `n`, with magnitude in the range `0.5 <= n < 1`, where `n` times 2 to the power of `exp` is equal to `value`.

## Related functions

`ldexp, modf`

## Example

```
#include <math.h>

double x = 3.141592;

double fraction;
int    exp;

fraction = frexp (x, &exp);
```

# fscanf

The `fscanf` function reads formatted input from a stream.

## Definition

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

## Purpose

This function reads input items controlled by the string pointed to by `format` from the file pointed to by `stream`. See the description of the function `scanf` for details on the `format` string. The function takes a variable number of arguments – the type and number of the arguments after the `format` argument is determined by the layout of the string pointed to by `format`.

## Returns

The `fscanf` function returns the number of input items assigned, or `EOF` if an input error occurred before any conversion was made.

## Related functions

`scanf, sscanf`

## Example

```
#include <stdio.h>

FILE *stream = fopen ("data", "r");

char name[20];
int  age;

while (!feof (stream) )
   { fscanf (stream, "Name %20s age %d\n", name, &age);
     if (age < 18) printf ("%s is under age\n", name);
   }
```

# fseek

The fseek function moves the file pointer associated with a stream.

## Definition

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

## Purpose

This function sets the file pointer (the location within the file at which the next input or output is performed) for the file associated with stream. The file pointer can be moved relative to three different positions depending on the value of whence as follows :-

| Whence | Meaning |
|---|---|
| SEEK_SET | relative to the start of the file |
| SEEK_CUR | relative to the current file position |
| SEEK_END | relative to the end of the file |

The macros SEEK_SET, SEEK_CUR and SEEK_END are defined in stdio.h.

The file's end-of-file indicator is cleared, and the effect of any previous call of ungetc is undone. If the stream was opened in update mode, the next operation after a call of fseek may be either input or output.

Note that for files opened in text mode, it is not meaningful to seek unless offset is zero or offset is a value previously returned by a call of ftell and whence is equal to SEEK_SET.

## Returns

The fseek function returns zero if successful. If the request was improper, such as an attempt to seek outside the bounds of the file, it returns non-zero, and errno will be set to indicate the error.

## Related functions

ftell, fgetpos, fsetpos

**Example**

```
#include <stdio.h>

FILE *stream = fopen ("data", "rb");

char last_ch;

fseek (stream, -1l, SEEK_END);

last_ch = fgetc (stream);
```

# fsetpos

The `fsetpos` function sets the file pointer associated with a stream.

## Definition

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

## Purpose

This function sets the file pointer (the location within the file at which the next input or output is performed) for the file associated with `stream` to position `pos`, which must be a value previously returned by `fgetpos`.

The file's end-of-file indicator is cleared, and the effect of any previous call of `ungetc` is undone. If the stream was opened in update mode, the next operation after a call of `fsetpos` may be either input or output.

## Returns

The `fsetpos` function returns zero if successful. If the request was invalid, it returns non-zero, and `errno` will be set to the value `EINVAL`.

## Related functions

`fgetpos, fseek, ftell`

## Example

```
#include <stdio.h>

FILE *stream;

fpos_t pos;

fgetpos (stream, &pos); /* get the position */
rewind (stream); /* move away */
fsetpos (stream, &pos); /* move back again */
```

# ftell

The `ftell` function returns the position of the file pointer associated with a stream.

### Definition

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

### Purpose

This function returns the position of the file pointer (the location within the file at which the next input or output is performed) for the file associated with `stream`. This value may then be used in a subsequent call of `fseek` to restore the file to that position.

### Returns

For binary files, `ftell` function returns the file position relative to the start of the file. However for either binary or text files, the value returned can be used by `fseek` to restore the file to that position. If an error occurs, it returns –1L, and `errno` will be set to indicate the error.

### Related functions

`fgetpos, fsetpos, fseek`

### Example

```
#include <stdio.h>

FILE *stream = fopen ("data", "rb");

long int number;

/* read some data */

number = ftell (stream);    /* bytes read so far */
```

# fwrite

The `fwrite` function writes data from an array to a stream.

## Definition

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

## Purpose

This function writes up to `nmemb` objects, each of `size` bytes, to the stream described by `stream` from the array whose base is given by `ptr`. If the stream was opened in text mode, any line-feeds output will be translated to carriage-return / line-feed pairs. This is unlikely to be what was intended.

## Returns

The `fwrite` function returns the number of objects successfully written – this will be less than `nmemb` only if a write error occurred. If an error occurred, the stream's error indicator and `errno` will be set.

## Related functions

`fopen, fread, read, write`

## Example

```
#include <stdio.h>

FILE *stream = fopen ("data", "rb");

double data[100];

int i;

for (i=0; i<100; i++)
    data[i] = sin ( ((double) i) / 100);

fwrite (data, sizeof (double), 100, stream);
```

# gemdos

The gemdos function makes an operating system TRAP #1 call. It is not part of the draft ANSI standard.

## Definition

```
#include <dos.h>
long int gemdos(int funcno, ...);
```

## Purpose

This function calls the operating system (gemdos) function whose number is given in the parameter funcno. The number, type and meaning of the other parameters depend on the function, and are documented in Atari technical information and a number of books about the Atari ST. A number of macros are declared in dos.h, one for each operating system function, which expand into calls of the gemdos function with appropriate parameters.

## Returns

The gemdos function returns the long value returned by the operating system TRAP #1 instruction. The meaning of this return value depends on the value of funcno, but frequently a negative value will indicate an error, corresponding to the positive error number with the same magnitude defined in errno.h.

## Related functions

bios, xbios

## Example

```
#include <dos.h>

/* Enter 68000 supervisor mode */
long int old_ssp;

oldssp = gemdos(0x20, 0l);
...
/* Now return to user mode */
gemdos(0x20, old_ssp);
```

# getc

The `getc` function reads a character from a stream.

## Definition

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

## Purpose

This function reads the next character from the file specified by `stream`. It is equivalent to `fgetc`, except that many C compilers (but not Prospero C) implement it as an unsafe macro (the argument `stream` may be evaluated twice, including possible side effects).

## Returns

The `getc` function returns the next character (converted to an `int`). If the stream is at end-of-file, the stream's end-of-file indicator will be set. If a read error occurs, the stream's error indicator is set. In both these cases, `getc` will return `EOF`.

## Related functions

`fgetc, getchar, ungetc, getch, getche`

## Example

```
#include <stdio.h>

/* append one file to the end of another */
FILE *f1, *f2;
char c;

f1 = fopen ("file1.dat", "r");
f2 = fopen ("file2.dat", "a");

while ((ch = getc(f1)) != EOF)
  putc(ch, f2);
```

# getch

The `getch` function reads a character from the keyboard. It is not part of the draft ANSI standard.

## Definition

```
#include <conio.h>
int getch (void);
```

## Purpose

This function reads a character from the keyboard, without echoing it to the screen.

## Returns

The `getch` function returns the character read (converted to an `int`).

## Related functions

`getchar, getche, ungetch`

## Example

```
#include <conio.h>

char ch;

fputs("Continue (y/n) ?", stdout);
do ch = toupper(getch());
   while (ch != 'Y' && ch != 'N');
if (ch == 'N')
  exit(1);
```

# getchar

The `getchar` function reads a character from a standard input.

## Definition

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

## Purpose

This function reads the next character from standard input. It is equivalent to `getc(stdin)`.

## Returns

The `getchar` function returns the next character (converted to an `int`). If `stdin` is at end-of-file, the end-of-file indicator will be set. If a read error occurs, the error indicator is set. In both these cases, `getchar` will return `EOF`.

## Related functions

`fgetc, getc, ungetc, getch, getche`

## Example

```
/* Append standard input to file "echo.log" */

#include <stdio.h>

FILE *echo;
char ch;

echo = fopen("echo.log", "a");
while ((ch = getchar()) != EOF)
  fputc(ch, echo);
```

# getche

The `getche` function reads a character from the keyboard. It is not part of the draft ANSI standard.

## Definition

```
#include <conio.h>
int getche (void);
```

## Purpose

This function reads a character from the keyboard and echoes it to the screen.

## Returns

The `getche` function returns the character read (converted to an `int`).

## Related functions

`getchar, getch, ungetch`

**Example**

```
/* a fairly simple line input routine */

#include <conio.h>

char ch, buffer[79];
int index = 0;

do {ch = getche();
    if (ch == '\b' && index > 0)
      { putchar(' ');       /* clear character */
        putchar('\b');      /* and backspace   */
        index--;
      }
    else if (ch != '\r')
    {  if (index == 78)     /* end of line     */
        { putchar('\a');    /* beep            */
          putchar('\b');    /* backspace       */
          putchar(' ');     /* clear the char  */
          putchar('\b');    /* and backspace   */
        }
       else if (isprint (ch))
          buffer[index++] = ch;
    }
    else
      buffer[index] = '\0';/* put in null terminator */
   } while (ch != '\r');
```

# getcwd

The `getcwd` function returns the pathname of the current working directory. It is not part of the draft ANSI standard.

## Definition

```
#include <direct.h>
char *getcwd (char *buffer, size_t size);
```

## Purpose

This function returns the full pathname (including drive letter) of the current working directory. If `buffer` is not `NULL`, up to `size` bytes (including a null terminator) of the pathname are stored in the object to which it points. If `buffer` is `NULL`, an array of `size` characters is allocated using `malloc`, and the pathname stored in it. This array can later be released by the program using `free`.

## Returns

The `getcwd` function returns a pointer to the buffer in which the pathname was written. If an error occurs, or the pathname is longer than `size - 1` characters, `NULL` is returned, and `errno` will be set to indicate the error.

## Related functions

`chdir, mkdir, rmdir`

## Example

```
#include <direct.h>
char *pathname;

pathname = getcwd (NULL, 80);
if (pathname != NULL)
{ printf ("Current directory is %s\n", pathname);
  free (pathname);
}
```

# getdfs

The `getdfs` function is used to determine the disk free space. It is not part of the draft ANSI standard.

## Definition

```
#include <direct.h>
void getdfs (int drive, struct DISKINFO *info);
```

## Purpose

This function examines the disk specified by `drive`, where 0 indicates the default drive, 1 means drive A and so on, and returns information about the amount of free space available in the structure pointed to by `info`. The structure `DISKINFO` is defined in `direct.h` as follows :-

```
struct DISKINFO
{   long free;        /* No. of free clusters       */
    long cpd;         /* No. of clusters per disk   */
    long bps;         /* No. of bytes per sector    */
    long spc; };      /* No. of sectors per cluster */
```

## Returns

There is no return value.

## Related functions

`getdisk, setdisk`

## Example

```
#include <direct.h>

struct DISKINFO dinfo;
long free, total;

getdfs(0, &dinfo);
free = dinfo.free * dinfo.bps * dinfo.spc;
total = dinfo.cpd * dinfo.bps * dinfo.spc;
printf ("%ld bytes free out of %ld\n", free, total);
```

# getdisk

The `getdisk` function returns the drive number of the current default drive. It is not part of the draft ANSI standard.

## Definition

```
#include <direct.h>
int getdisk (void);
```

## Purpose

This function is used to determine the current default drive.

## Returns

The `getdisk` function returns an integer indicating the current default drive, where 0 means drive A, 1 means drive B, and so on.

## Related functions

`setdisk, getcwd`

## Example

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

char command[256];

putchar(getdisk() + 'A');
putchar('>');
gets (command);
...
```

# getenv

The `getenv` function searches the program's environment data.

## Definition

```
#include <stdlib.h>
char *getenv (const char *name);
```

## Purpose

This function searches the program's environment list for a variable whose name is that given by the parameter `name`. All environment variables have upper case names, and so the parameter `name` must also be in upper case.

## Returns

The `getenv` function returns a pointer to the string assigned to the given environment variable. If the environment variable is not found, `NULL` is returned.

## Related functions

`spawn...`

## Example

```
#include <stdlib.h>

char * pathname;

pathname = getenv("TMP");
if (pathname != NULL)
  chdir(pathname);
workfile = fopen("temp.$$$", "w");
```

# gets

The `gets` function reads a string from standard input.

## Definition

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

## Purpose

This function reads input characters from the stream `stdin` into the array pointed to by `s`, until end-of-file is reached or a new-line character (which is not stored in the array) is read. A terminating null character is then written to the array. Note that unlike `fgets`, no checks are made that the number of characters read does not exceed the size of the array.

## Returns

The `gets` function returns `s` if successful. If end-of-file was encountered before any characters were read it returns `NULL`, and the contents of the array are not changed. If a read error occurs, `NULL` is returned and `errno` will be set to indicate the error. In this case the contents of the array will be indeterminate.

## Related functions

`fgets, getc, getchar`

## Example

```
#include <stdio.h>

char name[80];
puts("Please enter your name");
gets(name);
printf("Hello, %s\n", name);
```

# gmtime

The gmtime function converts a calendar time to Greenwich Mean Time.

## Definition

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
```

## Purpose

This function converts the time in the structure pointed to by timer into a broken down time, expressed as Greenwich Mean Time. The time pointed to by timer is assumed to be in the local time zone (normally, it will have been obtained using the time function), and the time difference between local time and GMT is determined by examining the environment variable TZ. The value of this variable should consist of three letters describing the standard time zone name, followed by a signed decimal number indicating the number of hours by which the local time precedes GMT, followed (if daylight saving time is in effect) by another three letter code describing the daylight saving time zone name.

## Returns

The gmtime function returns a pointer to a static structure containing the broken down time. This structure is overwritten by the next call of gmtime, asctime, ctime or localtime. If the TZ environment variable is not set, the gmtime function returns NULL, indicating that GMT is not available.

## Related functions

asctime, ctime, localtime, mktime, time, tzset

## Example

```
#include <time.h>

time_t t = time (NULL);
struct tm * GMT = gmtime(&t);

if (GMT != NULL)
  printf("GMT is %s\n", asctime(GMT));
```

# isalnum

The `isalnum` function tests whether a character is alphanumeric.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is either a letter or a digit. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isalnum` is `#undef`'d, a library function will be called.

## Returns

The `isalnum` function returns zero if `c` is neither a letter nor a digit, otherwise it returns non-zero.

## Related functions

`isalpha, isdigit, isxdigit`

## Example

```
#include <ctype.h>

char *name, *S;

/* Read a name consisting of letters and digits only */
while (isalnum(*s))
  *name++ = *s++;
*name = '\0';
```

# isalpha

The `isalpha` function tests whether a character is a letter.

### Definition

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

### Purpose

This function tests whether the character `c` (converted to an `int`) is either an upper case or a lower case letter. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isalpha` is `#undef`'d, a library function will be called.

### Returns

The `isalpha` function returns zero if `c` is neither an upper case nor a lower case letter, otherwise it returns non-zero.

### Related functions

`isalnum, isupper, islower`

### Example

```
#include <ctype.h>

char *s;

/* Check that a string starts with a letter */
if (!isalpha(*s))
  puts("Error - string must start with a letter");
```

# isascii

The `isascii` function tests whether a character is in the range 0 to 127. It is not part of the draft ANSI standard.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is in the range of the 7-bit ASCII code (0 to 127). It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isascii` is `#undef`'d, a library function will be called.

## Returns

The `isascii` function returns zero if `c` is not in the range 0 to 127, otherwise it returns non-zero.

## Example

```
#include <ctype.h>

char *s;
int  ch;

while (ch = *s++)
  if (!isascii(c))
    printf("Non-ASCII character %d encountered\n", c);
```

# iscntrl

The `iscntrl` function tests whether a character is a control character.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is a control character. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `iscntrl` is `#undef`'d, a library function will be called.

## Returns

The `iscntrl` function returns zero if `c` is not a control character, otherwise it returns non-zero.

## Related functions

`isgraph, isprint, ispunct`

## Example

```
#include <ctype.h>

int c;

do
{ c = getch();
  if (iscntrl (c))
    switch (c)
    { case 1: do_control_A();
              break;
      case 2: do_control_B();
              break
      ...
    }
} while (c != 3);  /* Until control C pressed */
```

# isdigit

The `isdigit` function tests whether a character is a decimal digit.

### Definition

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

### Purpose

This function tests whether the character `c` (converted to an `int`) is a digit. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isdigit` is `#undef`'d, a library function will be called.

### Returns

The `isdigit` function returns zero if `c` is not a digit, otherwise it returns non-zero.

### Related functions

`isalnum, isxdigit`

### Example

```
#include <ctype.h>

char *s;
int number;

/* Read and convert a decimal number */
number = 0;
while (isdigit(*s))
  number = number * 10 + *s++ - '0';
```

# isgraph

The `isalnum` function tests whether a character is a graphic character.

### Definition

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

### Purpose

This function tests whether the character `c` (converted to an `int`) is any printable character other than space. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isgraph` is `#undef`'d, a library function will be called.

### Returns

The `isgraph` function returns zero if `c` is not a printable character other than space, otherwise it returns non-zero.

### Related functions

`isprint, iscntrl`

### Example

```
#include <ctype.h>

/* Convert all non-printing characters and spaces in
   a string to underscores */

char *s;

while (*s)
  if (isgraph(*s))
     s++;
  else
     *s++ = '_';
```

# islower

The `islower` function tests whether a character is a lower case letter.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is a lower case letter. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `islower` is `#undef`'d, a library function will be called.

## Returns

The `islower` function returns zero if `c` is not a lower case letter, otherwise it returns non-zero.

## Related functions

`isalnum, isalpha, isupper`

## Example

```
#include <ctype.h>

char *s;
long int lower;

/* Count the lower case letters in a string */
lower = 0;
while (*s)
  if (islower(*s++))
    lower++;
```

# isprint

The `isprint` function tests whether a character is printable.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is any printable character (including space). It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isprint` is `#undef`'d, a library function will be called.

## Returns

The `isprint` function returns zero if `c` is not printable, otherwise it returns non-zero. All characters whose ASCII code is greater than or equal to that of the space character are considered printable, except for DEL (ASCII `0x7f`).

## Related functions

`isgraph, isspace, isascii`

## Example

```
#include <ctype.h>

/* print out a string, replacing unprintable characters
   with periods */

char *s, ch;

while ((ch = *s++) != '\0')
  putchar (isprint(ch) ? ch : '.');
```

# ispunct

The `ispunct` function tests whether a character is a punctuation character.

### Definition

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

### Purpose

This function tests whether the character `c` (converted to an `int`) is a punctuation character. This includes all printing characters except space or those for which `isalnum` is true. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `ispunct` is `#undef`'d, a library function will be called.

### Returns

The `ispunct` function returns zero if `c` is not a punctuation character, otherwise it returns non-zero.

### Related functions

`isalnum, isgraph, isprint`

### Example

```
#include <ctype.h>

/* break a string at the first punctuation character */

char *s;

while (*s && !ispunct (*s) )
  s++;
*s = '\0';
```

# isspace

The `isspace` function tests whether a character is white space.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is any white space character (i.e. space `' '`, form feed `'\f'`, newline `'\n'`, carriage return `'\r'`, horizontal tab `'\t'` or vertical tab `'\v'`). It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isspace` is `#undef`'d, a library function will be called.

## Returns

The `isspace` function returns zero if `c` is not a white space character, otherwise it returns non-zero.

## Related functions

`isgraph, ispunct`

## Example

```
#include <ctype.h>

/* skip leading white space */

char *s;

while (*s && isspace (*s) )
  s++;
```

# isupper

The `isupper` function tests whether a character is an upper case letter.

## Definition

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

## Purpose

This function tests whether the character `c` (converted to an `int`) is an upper case letter. It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isupper` is `#undef`'d, a library function will be called.

## Returns

The `isupper` function returns zero if `c` is not an upper case letter, otherwise it returns non-zero.

## Related functions

`isalnum, isalpha, islower`

## Example

```
#include <ctype.h>

/* Count the upper case letters in a string */

char *s;
long int upper;

upper = 0;
while (*s)
  if (isupper (*s++) )
    upper++;
```

# isxdigit

The `isxdigit` function tests whether a character is a hexadecimal digit.

### Definition

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

### Purpose

This function tests whether the character `c` (converted to an `int`) is a hexadecimal digit (i.e. one of the characters '0' to '9', 'A' to 'F' or 'a' to 'f'). It is declared as a macro in `ctype.h`, but if this is not `#included`, or if `isxdigit` is `#undef`'d, a library function will be called.

### Returns

The `isxdigit` function returns zero if `c` is not a hexadecimal digit, otherwise it returns non-zero.

### Related functions

`isdigit, isalpha, isalnum`

### Example

```
#include <ctype.h>

/* read and convert a hexadecimal number */

char *s;
unsigned long int number;

number = 0;
while isxdigit(*s)
  { int ch = *s++;
    if (isdigit (ch) )
      ch -= '0';
    else
      ch -= ( islower(ch) ? 'a' : 'A') - 10;
    number = number * 16 +ch;
  }
```

# itoa

The `itoa` function converts an integer into an ascii string. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
char *itoa (int value, char *string, int radix);
```

## Purpose

This function converts the integer `value` into ASCII characters in `string`, representing the value of the integer in the base `radix`. If `radix` is 10, and `value` is negative, the result will start with a minus sign, otherwise it is treated as unsigned. A terminating null character is always appended to the resulting string. The maximum number of characters which can be placed into the array pointed to by `string` is 17 (when `radix` is 2).

## Returns

The `itoa` function returns the value of `string`.

## Related functions

`ltoa, ultoa, sprintf, atoi, atol, strtol, strtoul`

## Example

```
#include <stdlib.h>

char str[17];
int n;

for (n = 0; n < 20; n++)
  printf (" %3d in binary is %10s \n", n,
                itoa (n, str, 2) );
```

# kbhit

The `kbhit` function tests whether a character is available in the keyboard buffer. It is not part of the draft ANSI standard.

### Definition

```
#include <conio.h>
int kbhit (void);
```

### Purpose

This function tests whether or not a key has been pressed since the one which was last read, and therefore whether or not a call of `getch` or `getche` would cause the program to pause.

### Returns

The `kbhit` function returns non-zero if a key has been pressed, otherwise zero.

### Related functions

`getch, getche`

### Example

```
#include <conio.h>

/* List a file, pausing if a key is pressed */
FILE *f = fopen("listfile", "r");

while (!feof(f))
{ fgets(buffer, f);
  fputs(buffer, stdout);
  if (kbhit())
  { getche();  /* Discard key which made us pause */
    while (getche() != ' ');
      /* Wait for a space before continuing */
  }
}
```

# labs

## Definition

```
#include <stdlib.h>
long int labs (long int j);
```

## Purpose

This function returns the absolute value of the long integer `j`.

## Returns

The `labs` function returns the absolute value of `j`.

## Related functions

`abs, fabs`

## Example

```
#include <stdlib.h>

main()
{  long int i;

   for (i = -49999; i < 49999; i += 10000)
       printf("%ld\n", labs(i));
}
```

produces as output

```
49999
39999
29999
19999
9999
1
10001
20001
30001
40001
```

# ldexp

The `ldexp` function multiplies a floating point value by a power of two.

### Definition

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

### Purpose

This function multiplies the floating point value `x` by 2 raised to the power of `exp`. This function is used internally by the library.

### Returns

The `ldexp` function returns `x` times 2 raised to the power of `exp`. If the result is too large to be represented as a double, a range error occurs. In this case, `errno` will be set to `ERANGE`, and the value `HUGE_VAL` will be returned.

### Related functions

`frexp`

### Example

```
#include <math.h>

printf ("1 Megabyte = %g bits\n", ldexp ( 8.0, 20) );
```

produces the output

```
1 Megabyte = 8.38861e+06 bits
```

# ldiv

The `ldiv` function computes the quotient and remainder of a long integer division.

### Definition

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

### Purpose

This function calculates the quotient and remainder of the division of `numer` by `denom`. The sign of the remainder is always the same as the sign of the quotient (unless the remainder is zero). If `denom` is zero, or `numer` is `MIN_LONG` and `denom` is `-1`, the result is undefined.

### Returns

The `ldiv` function returns a structure with two `long int` fields, `quot` and `rem`. The type `ldiv_t` is defined in `stdlib.h`.

### Related functions

`div`

### Example

```
#include <stdlib.h>

ldiv_t result;
long hours, minutes;

result = ldiv (minutes, 601);
hours = result.quot;
minutes = result.rem;
if (minutes < 0)
   { minutes += 60;  /* put into range */
     hours--;
   }
```

# localtime

The `localtime` function converts a calendar time to local time.

## Definition

```
#include <time.h>
struct tm *localtime (const time_t *timer);
```

## Purpose

This function converts the time in the structure pointed to by `timer` into a broken down time. The time pointed to by `timer` is assumed to be in the local time zone (normally, it will have been obtained using the `time` function).

## Returns

The `localtime` function returns a pointer to a static structure containing the broken down time. This structure is overwritten by the next call of `gmtime`, `asctime`, `ctime` or `localtime`.

## Related functions

`asctime, ctime, gmtime, mktime, time`

## Example

```
#include <time.h>

time_t t = time (NULL);

/* We could use ctime here ... */
printf("The time is %s\n", asctime(localtime(&t)));
```

# log

The log function calculates the natural logarithm of a value.

## Definition

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

## Purpose

This function calculates the natural logarithm of the floating point value x.

## Returns

The log function returns the logarithm. If x is zero, a range error occurs – errno is set to ERANGE, and -HUGE_VAL is returned. If x is negative, a domain error occurs – errno is set to EDOM, and the value 0.0 is returned.

## Related functions

exp, log10, pow

## Example

```
#include <math.h>

double cube_root(double x)
{ if (x<0)
        return -exp (log (-x) / 3);
  return exp (log (x) / 3);
}
```

# log10

The `log` function calculates the base ten logarithm of a value.

## Definition

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

## Purpose

This function calculates the base ten logarithm of the floating point value `x`.

## Returns

The `log10` function returns the base ten logarithm. If `x` is zero, a range error occurs – `errno` is set to `ERANGE`, and `-HUGE_VAL` is returned. If `x` is negative, a domain error occurs – `errno` is set to `EDOM`, and the value 0.0 is returned.

## Related functions

`exp, log, pow`

## Example

```
#include <math.h>

/* Calculate the exponent of a value - can't exceed
   integer range*/

double x;

int exponent =  floor (log10(x));
```

# longjmp

The `longjmp` function restores program execution to a previously saved state.

## Definition

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

## Purpose

This function restores the program execution environment to that saved in `jmp_buf` by a previous call of `setjmp`. The program counter, stack pointer, and registers are restored to their stored values, and execution continues as if the corresponding call of `setjmp` had returned the value specified by `val`.

This function is typically used to deal with error conditions, to avoid a long series of function returns.

## Returns

The `longjmp` function never returns to its caller, but instead causes the last call of `setjmp` with `jmp_buf` as its argument to return the value `val`. Note however that a return value of zero is reserved to mean indicate a return from a direct invocation of `setjmp`, so that if `val` is zero, the return value from `setjmp` will be one. If there was no such call of `setjmp`, or if the function containing the `setjmp` call is no longer active, the results will be undefined (almost certainly a system crash).

## Related functions

`setjmp`

**Example**

```
#include <setjmp.h>

jmp_buf error;

main()
{ int err = setjmp(error);
  if (err)
  { printf("Fatal error %d - terminating\n");
    exit(3);
  }
  process_file();
  ...
}

int process_file(void);
{ FILE *f = fopen("data.in", "r");
  if (f == NULL)
    longjmp(error, 5);
  ...
}
```

# lseek

The `lseek` function sets the file position of an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long lseek (int handle, long int offset, int whence);
```

## Purpose

This function sets the file pointer (the location within the file at which the next input or output is performed) for the unbuffered file associated with `handle`. The file pointer can be moved relative to three different positions depending on the value of `whence` as follows :-

| Whence | Meaning |
|---|---|
| `SEEK_SET` | relative to the start of the file |
| `SEEK_CUR` | relative to the current file position |
| `SEEK_END` | relative to the end of the file |

The macros `SEEK_SET`, `SEEK_CUR` and `SEEK_END` are defined in `stdio.h`.

Note that for files opened in text mode, it is not meaningful to seek unless `offset` is zero or `offset` is a value previously returned by a call of `tell` and `whence` is equal to `SEEK_SET`.

## Returns

The `lseek` function returns the new file position. If the request was improper, such as an attempt to seek outside the bounds of the file, it returns a negative value, and `errno` will be set to indicate the error.

## Related functions

`fseek, ftell, fgetpos, fsetpos, tell`

**Example**

```
#include <io.h>
#include <stdio.h>  /* For macro definitions */

/* Seek to end of file to determine size, then restore
   file pointer to original position */

long old_pos, size;
int handle;

old_pos = tell(handle);
size = lseek(handle, 0, SEEK_END);
lseek(handle, old_pos, SEEK_SET);
```

# ltoa

The `ltoa` function converts an integer into an ASCII string. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
char *ltoa (long int value, char *string, int radix);
```

## Purpose

This function converts the long integer `value` into ASCII characters in `string`, representing the value of the integer in the base `radix` (in the range 2 to 36). If `radix` is 10, and `value` is negative, the result will start with a minus sign, otherwise it is treated as unsigned. A terminating null character is always appended to the resulting string. The maximum number of characters which can be placed into the array pointed to by `string` is 33 (when `radix` is 2).

## Returns

The `ltoa` function returns the value of `string`.

## Related functions

`itoa, ultoa, sprintf, atoi, atol, strtol, strtoul`

## Example

```
#include <stdlib.h>

char str[33];
int base;

for (base = 2; base < 37; base++)
  printf(" 1000000 in base %2d is %s \n", base,
         ltoa(1000000, str, base));
```

# malloc

The `malloc` function allocates a block of memory dynamically.

### Definition

```
#include <stdlib.h>
void *malloc (size_t size);
```

### Purpose

This function allocates a block of memory of `size` bytes. The memory should be released when no longer required using `free`.

### Returns

The `malloc` function returns a pointer to the start of the allocated memory. If `size` is zero, or if insufficient memory is available, it returns `NULL`.

### Related functions

`calloc, free, realloc`

### Example

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

char * buffer = malloc(512);
if (buffer)
{ /* Use buffer */
  ...
  /* finished with buffer now, so release it */
  free (buffer);
}
else
  errno = ENOMEM; /* signal insufficient memory */
```

# memccpy

The `memccpy` function copies one block of memory to another, until a particular character is found. It is not part of the draft ANSI standard.

## Definition

```
#include <string.h>
void *memccpy(void *s1, const void *s2,
              int c, size_t n);
```

## Purpose

This function copies characters from the object pointed to by `s2` to that pointed to by `s1`, until either `n` characters have been copied, or the value of the last character copied was equal to `c` (converted to an `unsigned char`). If the objects overlap, the results are undefined.

## Returns

The `memccpy` function returns a pointer to the char immediately following `c`, in string `s1`, if copied, else `NULL`.

## Related functions

`memcpy, memmove, strcpy, strncpy`

## Example

```
#include <string.h>

char a[26];
int ch;

/* Copy all letters up to ch into a */
memccpy(a, "abcdefghijklmnopqrstuvwxyz", ch, 26);
```

# memchr

The memchr function locates the first occurrence of a character in a block of memory.

## Definition

```
#include <string.h>
void *memchr (const void *s, int c, size_t n);
```

## Purpose

This function searches the first n characters of the object pointed to by s for a character whose value is equal to c (converted to an unsigned char).

## Returns

The memchr function returns a pointer to the first occurrence of the character, or NULL if no occurrence is found.

## Related functions

strchr

## Example

```
#include <string.h>

char buffer[512];
char *nl_ptr;

/* Convert newlines in buffer to null characters
    (rather inefficiently) */
while ((nl_ptr = memchr (buffer, '\n', 511)) != NULL)
  *nl_ptr = '\0';
```

# memcmp

The memcmp function compares two blocks of memory.

## Definition

```
#include <string.h>
int memcmp (const void *s1, const void *s2, size_t n);
```

## Purpose

This function compares the first n characters of the objects pointed to by s1 and s2.

## Returns

The memcmp function returns an integer greater than, equal to or less than zero, according to whether the object pointed to by s1 is respectively greater than, equal to or less than that pointed to by s2. The comparison is made on the basis of the first character position in which they differ, treating the characters as unsigned.

## Related functions

memicmp, strcmp, stricmp, strncmp, strnicmp

## Example

```
#include <string.h>

int array1[20];
int array2[20];

/* Arrays cannot be compared using the == operator, but
   we can use memcmp to test if they are equal */

if (memcmp (array1, array2, 20*sizeof(int)) == 0)
  puts ("Arrays are equal");
else
  puts ("Arrays are not equal");
```

# memcpy

The memcpy function copies one block of memory to another.

## Definition

```
#include <string.h>
void *memcpy (void *s1, const void *s2, size_t n);
```

## Purpose

This function copies the first n bytes of the object pointed to s2 to the object pointed to by s1. The objects should not overlap, or the result will be undefined. Note that memmove can be used to copy objects which may overlap.

## Returns

The memcpy function returns a pointer to the destination object, s1.

## Related functions

memmove, strcpy, strncpy, strdup

## Example

```
#include <string.h>

#define linelength 80
char buffer[linelength];
char *screenptr;

/* Copy a line of a display to an array */

memcpy(buffer, screenptr, linelength);
```

# memicmp

The `memicmp` function compares two blocks of memory, ignoring case differences. It is not part of the draft ANSI standard.

## Definition

```
#include <string.h>
int memicmp (const void *s1, const void *s2, size_t n);
```

## Purpose

This function compares the first `n` characters of the objects pointed to by `s1` and `s2`, treating all letters as if they were upper case.

## Returns

The `memicmp` function returns an integer greater than, equal to or less than zero, according to whether the object pointed to by `s1` is respectively greater than, equal to or less than that pointed to by `s2`. The comparison is made on the basis of the first character position in which they differ, treating the characters as unsigned, and treating all letters as if they were upper case.

## Related functions

`memcmp, strcmp, stricmp, strncmp, strnicmp`

## Example

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

char *s1, *s2;

s1 = "Hello World";
s2 = "HELLO world";
assert (memicmp (s1, s2, 11) == 0);

/* Assertion should succeed */
```

# memmove

The memmove function copies one block of memory to another.

## Definition

```
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
```

## Purpose

This function copies the first n bytes of the object pointed to s2 to the object pointed to by s1. Unlike the memcpy function, memmove can be used to copy objects which overlap, and will behave correctly. However, for objects which are known not to overlap, memcpy is more efficient and faster.

## Returns

The memmove function returns a pointer to the destination object, s1.

## Related functions

memcpy, strcpy, strncpy, strdup

## Example

```
#include <string.h>

#define linelength 80
#define lines 25
char *screenptr;

/* Reverse scroll a character mapped display */

memmove (screenptr + linelength, screenptr,
          (lines-1) * linelength);
memset (screenptr, ' ', linelength);
```

# memset

The `memset` function fills a block of memory with a character.

## Definition

```
#include <string.h>
void *memset (void *s, int c, size_t n);
```

## Purpose

This function sets the first `n` characters of the object pointed to by `s` to the value `c` (converted to an `unsigned char`).

## Returns

The `memset` function returns a pointer to the destination object, `s`.

## Related functions

`strset`

## Example

```
#include <string.h>

/* Fill an array with zeros, quickly */

int s[100];

memset(s, 0, 100*sizeof(int));
```

# mkdir

The `mkdir` function creates a subdirectory. It is not part of the draft ANSI standard.

### Definition

```
#include <direct.h>
int mkdir (const char *pathname);
```

### Purpose

This function creates a new directory with the name specified by `pathname`. This may be either absolute or relative to the current default directory. The new directory will be created on the current drive.

Note that to include a backslash character in a string literal, two backslashes must be used, as the backslash character is used to introduce escape sequences. A forward slash may be used in place of the backslash character in `pathname` – it will be interpreted as if it were a backslash by the `mkdir` function.

### Returns

The `mkdir` function returns zero if the operation is successful. If the specified pathname cannot be created, a non-zero value is returned, and `errno` will be set to indicate the error (attempting to create a directory that already exists gives the error `EACCES`).

### Related functions

`chdir, rmdir`

### Example

```
#include <direct.h>

mkdir ("\\new");   /* new directory in root   */
mkdir ("new");     /* new directory in current  */
```

# mktime

The `mktime` function converts a broken down time to an encoded one.

## Definition

```
#include <time.h>
time_t mktime (struct tm *timeptr);
```

## Purpose

This function converts the broken down time in the structure pointed to by `timeptr` into a calendar time, using the same encoding as that returned by the time function. The values in the broken down time are adjusted in order to bring them all into their standard ranges, and the day of the week (`timeptr->tm_wday`) and of the year (`timeptr->tm_yday`) are recalculated from the other values in the structure. The broken down time structure `tm` is defined in the `time.h` file.

## Returns

The `mktime` function returns the encoded time, provided the (adjusted) year is in the range 1980 to 2099. Otherwise the given time can not be represented in the code used by the `time` function, and it returns `(time_t) -1`.

## Related functions

`localtime, time, gmtime`

## Example

```
#include <time.h>

/* What will the encoded time be in an hour ? */

time_t now, then;
struct tm *timeptr;

time(now);                  /* Get current time */
timeptr = localtime(&now)   /* break it down */
timeptr->tm_hour++;         /* add an hour */
then = mktime(timeptr)      /* and recode the time */
```

# modf

The `modf` function breaks a floating point value into integral and fractional parts.

## Definition

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

## Purpose

This function divides the floating point value `value` into its integral and fractional parts, each (if non-zero) having the same sign as `value`. The integral part is stored in the object pointed to by `iptr`.

## Returns

The `modf` function returns the fractional part of `value`.

## Related functions

`frexp, fmod, floor, ceil`

## Example

```
#include <math.h>

/* test whether a number is integral */

double integral, value;

if (modf(value, &integral) == 0.0)
  puts("Value is integral");
else
  puts("Value is not integral");
```

# open

The `open` function opens an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
int open (const char *filename, unsigned int access,
          int prot);
```

## Purpose

This function opens the file whose name is specified by filename. The mode in which it is opened is governed by the value of the `access` parameter, which is specified by combining one or more of the following values (defined in `fcntl.h`) using the bitwise OR operator `|` :-

| | |
|---|---|
| `O_RDONLY` | The file is to be opened for reading only |
| `O_WRONLY` | The file is to be opened for writing only |
| `O_RDWR` | The file is to be opened for both reading and writing. Exactly one of the above must appear. |
| `O_NDELAY` | Not used, included only for compatibility |
| `O_APPEND` | All write accesses start at end of file |
| `O_CREAT` | The file is to be created if it does not exist |
| `O_TRUNC` | The file is to be truncated to zero length if it exists |
| `O_EXCL` | If `O_CREAT` is specified, the file must not already exist |
| `O_TEXT` | The file is to be opened in text mode |

The `prot` parameter is only meaningful if `O_CREAT` is specified, and gives the file protection mode which the newly created file will be given. The possible values are obtained by combining (using the bitwise OR operator) one or both of the following values (also defined in `fcntl.h` ) :-

| | |
|---|---|
| `S_IWRITE` | The file is to be created with write permission |
| `S_IREAD` | The file is to be created with read permission |

Under GEMDOS, all files have read permission, so the modes `S_IWRITE` and `S_IWRITE | S_IREAD` are identical. However, it is sensible to specify `S_IREAD` if read permission is required, to assist portability to other operating systems.

### Returns

The `open` function returns the handle of the file. If an error occurs, –1 is returned, and `errno` will be set to indicate the error.

### Related functions

`access, chmod, close, creat, dup, dup2, fopen`

### Example

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

int handle;

/* create a new file */

handle = open ("newfile.dat",
               O_WRONLY | O_CREAT | O_EXCL,
               S_IREAD | S_IWRITE);
if (handle == -1)
{ if (errno == EEXIST)
    puts("File already exists");
  else
    perror("Error creating file newfile.dat");
  exit(3);
}
```

# perror

The `perror` function prints an error message.

### Definition

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

### Purpose

This function prints an error message to the standard error stream. If `s` is not `NULL`, the string it points to is printed first, followed by a colon and a space. An error string describing the error number in `errno` is then printed – this is the same string as returned by `strerror` with argument `errno`. Finally, a new-line character is output.

### Returns

There is no return value.

### Related functions

`strerror`

### Example

```
#include <stdio.h>

FILE *stream;

stream = fopen("myfile.dat", "r");
if (stream == NULL)
{  perror("Unable to open myfile.dat");
   exit(3);
}
```

would output (if file cannot be found):-

```
Unable to open myfile.dat: File not found
```

# pow

The pow function raises a floating point value to a power.

## Definition

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

## Purpose

This function computes x raised to the power y.

## Returns

The pow function returns x raised to the power of y. If the result is too large to be represented as a double, a range error occurs. In this case, errno will be set to ERANGE, and the value HUGE_VAL will be returned. If x is zero and y is less than or equal to zero, or if x is negative and y is not an integer, a domain error occurs. In this case, errno will be set to EDOM, and the value 0.0 will be returned.

## Related functions

frexp

## Example

```
#include <math.h>
#include <sttdef.h>     /* to define errno */
double d;

errno = 0;
for (d = 1.0; errno == 0; d *= 4.0)
  printf("%g\n", pow (2.0, d));
```

will produce as output:-

```
2
16
65536
1.84467e+19
1.15792e+77
1.79769e+308
```

# printf

The `printf` function writes formatted output to standard output.

## Definition

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

## Purpose

This function outputs characters to standard output, under the control of the format string `format`. Additional arguments may be passed, and conversion specifiers in the format string (described in detail below) will cause these arguments to be converted and their ASCII representations output.

The format string consists of a number of directives, which take two forms. The simple form of directive is an ordinary character (other than %), which causes that character to be written to the output unmodified.

Conversion specifications start with a % character, followed in sequence by :-

1 Zero or more of the flag characters "+", "–", "0", " " (space) or "#", which modify the effect of the conversion as follows :-

"+" For signed numeric conversions, a positive number will start with a + (negative numbers always have a sign).

" " For signed numeric conversions, a positive number will start with a space. If both the space and "+" flags appear, the space is ignored.

"0" For numeric conversions, zero characters are used to pad to the field width (after any sign or indication of base). If both the "0" and "–" flags appear, the "0" is ignored.

"–" If a field width is specified (see below), the output will be left justified in the field. If this flag does not appear, the output is right justified.

"#" An alternate form of the conversion is to be used. The effect on each type of conversion is described individually.

2 An optional decimal integer which gives the minimum field width. If the output resulting from the conversion occupies less than this number of characters, it will be padded on the left (or right if the left-justify flag (–) was given). The padding is normally with spaces, but if the zero flag was given, left padding is performed with zeros. The decimal integer may be replaced by an asterisk, in which case the next argument in the argument list is a signed int whose value is to be used instead. A negative argument is treated as if it was a left-justify flag followed by the corresponding positive argument.

3 An optional precision. This is introduced by a decimal point, followed by a decimal integer. The decimal integer may be replaced by an asterisk, meaning that the next argument in the argument list is a signed int whose value is to be used instead. A negative argument is treated as if it were missing.

4 An optional length specifier. This may be an "h", specifying that an integer conversion is to treat the next argument as being `short`, an "l", specifying that an integer conversion is to treat the next argument as being `long`, or an "L", specifying that a floating point conversion is to treat the next argument as being `long double`. If no length is specified, integers are assumed to be plain, and floating point values are assumed to be `double`. (Note that values of type `float` will be converted to `double` before being passed).

5 A character specifying the conversion to be performed. These are described in detail below.

"d", "i" An integer argument is converted to signed decimal notation. For all integer conversions, the precision specifies the minimum number of characters which appear, and leading zeros will be added if necessary. If no precision is specified, 1 is assumed. Note that outputting a zero value with zero precision results in no output.

"o" An integer argument is converted to unsigned octal notation. If the "#" (alternate form) flag was given, the precision will be increased if necessary to force the first character to be a zero.

"u" An integer argument is converted to unsigned decimal notation.

"x", "X" An integer argument is converted to unsigned hexadecimal notation. Lower case letters are used for "x" conversion, and upper case for "X". If the "#" (alternate form) flag was given, and the value is not zero, the characters "0x" or "0X" will be prepended to the output.

"e","E" A floating point value is converted to scientific decimal notation, of the form [-]d.dddddde±dd. The precision specifies the number of characters to appear after the decimal point – if no precision is given, it is assumed to be 6. If the precision is zero, no decimal point will be output, unless the "#" (alternate form) flag was specified. The digit before the decimal point is not zero unless the value is zero. The exponent sign and at least two exponent digits are always output. The value is rounded to the appropriate number of places. The "E" form differs from the "e" form only in that the letter "E" is used to introduce the exponent rather than the letter "e".

"f" A floating point value is converted to fixed point decimal notation. The precision specifies the number of characters to appear after the decimal point – if no precision is given, it is assumed to be 6. If the precision is zero, no decimal point will be output, unless the "#" (alternate form) flag was specified. At least one digit will always be output before the decimal point. The value is rounded to the appropriate number of places.

"g","G" A floating point value is converted using either the "e" (or "E" for "G" conversion) or "f" style, depending on the magnitude of the value. The precision specifies the number of significant digits – if no precision is given, it is assumed to be 6, and if a precision of zero is given, it is treated as one. The "e" form is used if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision, otherwise the "f" form is used. Unless the "#" (alternate form) flag is specified, trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit.

"c" An argument of type `int` is converted to an `unsigned char`, and the corresponding character is output.

"s" An argument of type `char *` is consumed. Characters from the string it points to are output, until either the number of characters output is equal to the precision (if specified), or the end of the string is reached. The terminating null is not written.

"p" An argument of type `void *` is converted to an unsigned long integer, and output in the same form as for the "x" conversion, except that the characters "0x" are prepended to the result whether or not the "#" flag was specified.

"n" No output is performed. An argument of type pointer to integer is consumed, and the number of characters so far written by this call of `printf` is stored in the unsigned integer to which it points. The size of the integer is determined by the length flags described earlier.

"%" A percent character is written. No argument is consumed. No other options may precede the %.

If an invalid conversion specification is encountered, the rest of the format string will be output without attempting to recognize conversion specifiers.

### Returns

The `printf` function returns the number of characters output. If an output error occurred, a negative value is returned, and `errno` will be set to indicate the error.

### Related functions

`fprintf, sprintf, vfprintf, vprintf, vsprintf, scanf`

### Example

```
#include <stdio.h>

int c = 'A';
double d = 3.14e-25;
char *s = "Hello";

printf ("c has value %d, and as a character is %c\n",
                    c, (char) c);
printf ("d is %.2g to 2 s.f. but %.2e to 2 d.p.\n",
                    d, d);
printf ("%10s %.2s\n", s, s);
```

will produce:-

```
c has value 65, and as a character is A
d is 3.1 to 2.s.f but %.2e to 2 d.p.
     Hello He
```

# putc

The putc function writes a character to a stream.

## Definition

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

## Purpose

This function writes the character c to the stream described by stream. It is equivalent to fputc, except that many C compilers (but not Prospero C) implement it as an unsafe macro (the argument stream may be evaluated twice, including possible side effects).

## Returns

The putc function returns the character written. If a write error occurs, EOF is returned and the file error flag and errno will be set.

This function is declared as a macro in stdio.h, but if this is not #included, or if putc is #undef'd, a library function will be called.

## Related functions

fputc, putchar, fputs

## Example

```
#include <stdio.h>
int errors;
FILE *logfile;

fprintf(logfile, "%d error", errors);
if (errors != 1)
  putc('s', logfile);
fprintf(logfile, " in Pass 1\n");
```

# putch

The `putch` function writes a character to the console. It is not part of the draft ANSI standard.

### Definition

```
#include <conio.h>
int putch (int c);
```

### Purpose

This function writes the character `c` (converted to a `char`) to the console.

### Returns

The `putch` function returns the character written.

### Related functions

`getch, getche, ungetch`

### Example

```
#include <conio.h>

char *p;

for (p = "Hello"; *p; p++)
  putch (*p);
```

# putchar

The `putchar` function writes a character to standard output.

## Definition

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

## Purpose

This function writes the character `c` to the standard output stream. It is equivalent to `putc` with `stdout` as the second argument.

This function is declared as a macro in `stdio.h`, but if this is not `#included`, or if `putchar` is `#undef`'d, a library function will be called.

## Returns

The `putchar` function returns the character written. If a write error occurs, `EOF` is returned and the file error flag and `errno` will be set.

## Related functions

`fputc, putc, fputs`

## Example

```
#include <stdio.h>
int errors;

printf("%d error", errors);
if (errors != 1)
  putchar('s');
printf(" in Pass 1\n");
```

# puts

The `puts` function writes a string of characters to standard output.

## Definition

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

## Purpose

This function writes the string of characters pointed to by `s` to the standard output stream, up to but not including the terminating null byte. A new-line character is then written to standard output.

## Returns

The `puts` function returns zero if successful. If a write error occurs, a non-zero value is returned, and the stream's error indicator and `errno` will be set.

## Related functions

`fputs, fputc, putc`

## Example

```
#include <stdio.h>

/* Output the sign-on message */

puts("Prospero C Cross referencer version mg 1.1");
puts("Copyright (C) 1988 Prospero Software");
```

# qsort

The `qsort` function sorts an array of objects.

## Definition

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *));
```

## Purpose

This function sorts `nmemb` members of the array pointed to by `base` into ascending order, using the Quicksort algorithm. The size of each object in the array is specified by the parameter `size`, and the comparisons are done on the basis of the values returned by the function pointed to by the parameter `compar`. This function is called by `qsort` with pointers to two members of the array as arguments, and should return an integer less than, equal to or greater than zero, according to whether the member pointed to by the first argument is considered respectively less than, equal to or greater than that pointed to by the second. If two members are equal according to the `compar` function, the order in which they appear in the sorted array will not necessarily be the same as that in the original.

## Returns

There is no return value.

## Related functions

`bsearch`

**Example**

```
#include <stdlib.h>

struct person {char surname[20];
               char first_name[20];
               int  year_of_birth; };

int compare_names (const void *v1, const void *v2)
{ struct person *p1 = (struct person *) v1,
                *p2 = (struct person *) v2;
  int order;

  order = strcmp(p1->surname, p2->surname);
  if (order == 0)
  { /* Surnames are the same - compare first names */
    order = strcmp(p1->first_name, p2->first_name);
    if (order == 0)
      /* First names are the same too - compare ages */
      order = p1->year_of_birth - p2->year_of_birth;
  }
  return order;
}

main()
{ struct person people[100];
  /* Set up info about 100 people */
  ...
  /* Now sort into ascending order of surname */
  qsort(people, 100,  sizeof(struct person),
        compare_names);

  /* Now print out the data etc */
  ...
}
```

# raise

The `raise` function causes an exception to be generated.

### Definition

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

### Purpose

This function sends the signal specified by `sig` to the executing program. This should be one of the values defined in the `signal.h` header file. The `signal` function can be used to control what the effect of each exception is.

### Returns

The `raise` function returns zero if successful, otherwise non-zero.

### Related functions

`signal`

### Example

```
#include <signal.h>

/* Raise a floating point exception */

raise (SIG_FPE);
```

# rand

The `rand` function generates a random number.

## Definition

```
#include <stdlib.h>
int rand (void);
```

## Purpose

This function calculates the next number in a pseudo-random sequence, in the range 0 to `RAND_MAX`. `RAND_MAX` is defined in `stdlib.h`, and in Prospero C is equal to 32767. The starting point of the sequence can be set using the `srand` function – at program startup the seed will be zero.

## Returns

The `rand` function returns the pseudo-random number.

## Related functions

`srand`

## Example

```
#include <stdlib.h>

if (rand() > 0x3fff)
  puts("Heads");
else
  puts("Tails");
```

# read

The `read` function reads data from an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long int read (int handle, void *buffer,
               long int length);
```

## Purpose

This function reads up to `length` bytes, from the file whose handle is given by `handle` into the object pointed to by `buffer`. If the file was opened in text mode, carriage returns in the input will be removed, and will not be counted towards the number of bytes read.

## Returns

The `read` function returns the number of bytes read into `buffer` – this may be less than `length` if end-of-file was encountered. If an error occurred, it returns `-1L`, and `errno` will be set to indicate the error.

## Related functions

`open, fread, fwrite, _read, write`

**Example**

```
#include <io.h>
int handle;
char screenimage [32768];

handle = open("screen1.dmp", O_RDWR, 0);
if (handle == 0)
  perror("Can't open dump file");
else
{ unsigned int ret = read (handle, screenimage, 32768);
  if (ret != 32768)
  { if (ret == 0xffff)
      perror("Error reading dump file");
    else
      fputs("Dump file format error", stderr);
    exit(3);
  }
  /* Now process screen image */
  ...
}
```

# _read

The _read function reads data from an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long int _read (int handle, void *buffer,
                long int length);
```

## Purpose

This function reads up to length bytes, from the file whose handle is given by handle into the object pointed to by buffer. This function makes a direct call to GEMDOS, without any translation of carriage returns for text files, and without setting errno if errors are detected.

This function is defined as a macro in io.h, but if io.h is not #included, or if _read is #undef'd, a library function will be called.

## Returns

The _read function returns the number of bytes read into buffer – this may be less than length if end-of-file was encountered. If an error occurred, it returns a negative GEMDOS error code, equal in magnitude to one of the positive error codes defined in errno.h.

## Related functions

open, fread, fwrite, read, write, _write

**Example**

```
#include <io.h>
int handle1, handle2;
char *fileimage;
long bytes;

/* Duplicate a file */
handle1 = open("file1", O_RDONLY, 0);
handle2 = open("file2", O_WRONLY | O_CREAT | O_TRUNC,
               S_IREAD | S_IWRITE);

bytes = filelength(handle1);
fileimage = malloc (bytes);
if (fileimage != NULL)
{ _read(handle1, fileimage, bytes);
  _write(handle2, fileimage, bytes);
}
else puts("Insufficient memory");
```

# realloc

The `realloc` function changes the size of an allocated memory block.

## Definition

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

## Purpose

If `ptr` is `NULL`, the realloc function behaves like the `malloc` function with the parameter `size`. Otherwise, `ptr` should point to a memory block previously allocated using `malloc`, and the size of this block will be altered to `size` bytes. The first n bytes of the object will be unchanged, where n is the lesser of the old and new sizes. Changing the size to zero will result in the block being freed.

## Returns

The `realloc` function returns a pointer to the start of the resized memory block. If the object is being increased in size, this pointer may not be the same as the original pointer, and any copies of the original pointer will no longer be valid. If insufficient memory is available, `realloc` returns `NULL`, and the original block pointed to by `ptr` is unchanged. If size is zero, it returns `NULL`, and the original block pointed to by `ptr` is freed.

## Related functions

`calloc, free, malloc`

## Example

```
#include <stdlib.h>
#include <string.h>
char *save_name, *name;

save_name = strdup(name);  /* save a copy of name */
/* Get next name */
...
/* Save copy of name, reusing space from last */
save_name = realloc(save_name, strlen(name) + 1);
strcpy(save_name, name);
```

# remove

The `remove` function deletes a file from disk.

### Definition

```
#include <stdio.h>
int remove (const char *filename);
```

### Purpose

This function deletes the file specified by the `filename` parameter.

### Returns

The `remove` function returns zero if successful. Otherwise, it returns non-zero, and `errno` will be set to indicate the error.

### Related functions

`rename, tmpfile`

### Example

```
#include <stdio.h>

if (remove("workfile.$$$"))
   perror("Unable to remove workfile");
```

# rename

The rename function renames a file on disk.

## Definition

```
#include <stdio.h>
int rename (const char *old, const char *new);
```

## Purpose

This function renames the file whose name is given by old to the name given by new.

## Returns

The rename function returns zero if successful. If an error occurred, it returns non-zero, and errno will be set to indicate the error. Possible causes of failure include a file called new already existing, a file called old not being found, or old and new referring to different drives or directories. If the named file is currently open, any stream associated with the old name will now be associated with the new name, and no error shall result.

## Related functions

remove

## Example

```
#include <stdio.h>

if (rename("editfile.c", "editfile.bak"))
  perror("Unable to create backup file");
```

# rewind

The `rewind` function restores a stream's position to the start of a file.

## Definition

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

## Purpose

This function sets the file position of the file associated with stream to the start of the file, and clears the stream's error and end-of-file indicators. If the file was opened in update mode, the next operation after a call of rewind may be either input or output.

## Returns

There is no return value.

## Related functions

`fseek, fsetpos`

## Example

```
#include <stdio.h>

FILE *stream;

/* create a file */
stream = fopen("myfile","w+b");

/* write lots of data to it */
...
/* Go back to the start */
rewind(stream);
/* Now we can read the data back */
```

# rmdir

The `rmdir` function removes a subdirectory. It is not part of the draft ANSI standard.

## Definition

```
#include <direct.h>
int rmdir (const char *pathname);
```

## Purpose

This function removes the directory with the name specified by `pathname`. This may be either absolute or relative to the current default directory. The directory can only be removed from the current drive.

Note that to include a backslash character in a string literal, two backslashes must be used, as the backslash character is used to introduce escape sequences. A forward slash may be used in place of the backslash character in `pathname` – this will be interpreted as if tit was a backslash by the `rmdir` function.

## Returns

The `rmdir` function returns zero if the operation is successful. If the specified directory cannot be removed, a non-zero value is returned, and `errno` will be set to indicate the error. Note that a directory which is not empty, or which is the current directory, cannot be removed.

## Related functions

`chdir, mkdir`

## Example

```
#include <direct.h>

rmdir ("\\old");   /* delete directory in root   */
rmdir ("old");     /* delete directory in current  */
```

# scanf

The `scanf` function reads formatted input from standard input.

## Definition

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

## Purpose

This function reads formatted input from standard input, under the control of the format string `format`. Additional pointer arguments may be passed, and conversion specifiers in the format string (described in detail below) will cause input items to be to be converted and their values assigned to the objects pointed to by these arguments.

The format string consists of a number of directives, each of which specifies the form of the expected input. The directives are processed in turn, until one fails – this can be due to no more input being available (an input failure) or the input characters being inappropriate (a matching error). There are three types of directive :-

A directive which consists of white space (as defined by the `isspace` function) causes input to be read up to but not including the first non-space character, or until no more characters are available.

A directive which consists of an ordinary character (other than %) causes the next input character to be read, and fails with a matching error if the character is not the same as that forming the directive.

Directives which start with % are conversion specifiers. After the % character, the following appear in sequence :-.

1 An optional *, indicating that the value resulting from the conversion is not to be assigned to an object.

2 An optional decimal integer which gives the maximum field width. No more than this number of input characters will be read.

3 An optional character indicating the size of the object which is to receive the converted value. For integer conversions, an "h" length specifier indicates that the corresponding argument is a pointer to a `short int`, while an "l" indicates it is a pointer to `long int`; for floating point conversions, an "l" length specifier indicates that the corresponding argument is a pointer to a `double`, while an "L" indicates it is a pointer to `long double`. If no length specifier is given, integer conversions assume the pointer is to a plain `int`, and floating point conversions that it is to a `float`.

4 A character specifying the conversion to be performed. These are described in detail below. Each conversion is performed in the following steps (except where indicated in the individual descriptions) :–

White space characters in the input are skipped.

Characters are read from the input until the maximum field width (if given) is reached, no more characters are available, or the next input character (which is left unread) is not valid in the corresponding position of an item of the specified form. If no characters can be read (other than the initial white space), the directive fails. This is a matching failure, unless a read error prevented characters being read from standard input, in which case it is an input failure.

The characters read above are converted into a value of the appropriate type. If they are not a matching sequence, a match failure occurs. If the vale is too large to be represented, the resulting value will be undefined.

Unless the assignment suppressing flag * was given, the result of the conversion is placed in the object pointed to by the next argument after `format` which has not yet been used. If this does not point to an object of appropriate type, the result will be undefined.

The valid conversion specification characters are as follows :–

"d" Matches an optionally signed decimal integer, in the same format as for the `strtol` function with `base` 10.

"i" Matches an optionally signed integer, in the same format as for the `strtol` function with `base` 0. The radix of the integer is determined by the initial characters.

"o" Matches an unsigned octal integer, in the same format as for the `strtoul` function with `base` 8.

"u" Matches an unsigned integer, in the same format as for the `strtoul` function with `base` 10.

"x", "X" Matches an unsigned hexadecimal integer, in the same format as for the `strtoul` function with `base` 16.

"e","E","f","g","G"
Matches an optionally signed floating point value, in the same format as for the `strtod` function.

"c" Matches a sequence of characters whose length is as specified by the field width, or 1 if no field width is given. Initial white-space characters are not skipped. The corresponding argument should be a pointer to the start of an array of characters large enough to hold the sequence. No null character is appended to the array.

"s" Matches a sequence of non-whitespace characters. The argument should be a pointer to the start of an array of characters large enough to hold the sequence, and a null character which is appended to the sequence automatically.

"[" Matches a non-empty sequence of characters from a set of characters defined by the format string as described below. Initial white-space characters are not skipped. The corresponding argument should be a pointer to the start of an array of characters large enough to hold the sequence, plus a null character which is appended automatically.

The characters which follow the "[" character in the format string, up to and including the matching "]" character, form part of the directive, and the characters between the brackets (the scanset) indicate those characters which will be matched by the directive, unless the first character after the "[" is a "^", in which case the directive will match any character except those in the scanset following the "^". As a special case, if the "[" is followed immediately by a "]" or by "^]", the "]" character is considered to be part of the scanset, and the next "]" will be considered to mark the end.

"p" Matches an argument in the same form as that output by `printf` using the %p directive (a hexadecimal constant starting with the characters "0x"). The value is converted to a pointer to `void`, and the corresponding argument should be a pointer to pointer to `void`. If the value read in is not a value written out using `printf` earlier in the same invocation of the program, it is likely to be meaningless and lead to undefined behavior.

"n" No input characters are read, and white space is not skipped. The corresponding argument should be of type pointer to integer, and the number of input characters so far read by this call of `scanf` is stored

in the unsigned integer to which it points. The size of the integer is determined by the length specifier described earlier. A %n directive is not included in the count of successful assignments returned by `scanf`.

"%" Matches a % character. No whitespace is skipped, and no argument is consumed. The complete conversion specifier should be %%, without any flags or fieldwidth.

If an invalid conversion specification is encountered, the rest of the format string will be ignored.

### Returns

The `scanf` function returns the number of input items assigned, which may be less than the number provided for or zero if a match failure occurs. If an input failure occurs before any conversion, `EOF` is returned.

### Related functions

`fscanf, sscanf, printf`

### Example

```
#include <stdio.h>

char name[20];
int  age;

printf ("Enter your name and age\n");
scanf ("%19s%*[^ ,\n]%d\n", name, &age);
```

will accept

```
Fred 22
Bloggs,23
```

or the name and age on separate lines.

# setbuf

The setbuf function allows the buffer for a stream to be specified.

## Definition

```
#include <stdio.h>
void setbuf (FILE *stream, char *buf);
```

## Purpose

This function specifies that the array pointed to by buf is to be used as the buffer for input and output to the file specified by stream. The size of the array should be at least BUFSIZ bytes (defined in stdio.h), and must remain in existence at least as long as the file is open. If buf is NULL, input and output to the file stream will be unbuffered. The setbuf function must be called after the file has been opened, but before any input or output to the file has taken place.

This function is equivalent to the setvbuf function with mode equal to _IOFBF and size equal to BUFSIZ, or if buf is NULL, with mode equal to _IONBF.

## Returns

There is no return value.

## Related functions

setvbuf, fopen

## Example

```
#include <stdio.h>

FILE * stream = fopen( "data", "wb+");
void * buffer = malloc ( BUFSIZ);

setbuf (stream, buffer );

/* file output is now buffered */
   ...
free (buffer);
```

# setdisk

The `setdisk` function sets the default drive. It is not part of the draft ANSI standard.

### Definition

```
#include <direct.h>
int setdisk (int drive);
```

### Purpose

This function is used to set the default drive to that specified by the parameter drive, where 0 means drive A, 1 means drive B, and so on.

### Returns

The `setdisk` function returns the previous default drive, in the same format. A negative value indicates an error, and `errno` will be set. Note however that GEMDOS does not give an error if a non-existent drive is nominated, but will instead fail on the next attempt to open or create a file. The drives that are available can be found using the function `drivemap`.

### Related functions

`chdir, drivemap, getdisk`

### Example

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

char drive;             /* The drive letter */

if (isalpha(drive))
  setdisk (toupper (drive) - 'A');
```

# setjmp

The setjmp function saves the program execution state.

## Definition

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

## Purpose

This function saves the current program execution environment in jmp_buf. A subsequent call of longjmp with jmp_buf as its first parameter will restore the program counter, stack pointer, and registers to the values stored at the time when setjmp was called, causing execution to continue as if the call of setjmp had returned.

This function is typically used to allow an error occurring in deeply nested functions to return quickly and simply to a high level routine where the error can be reported and suitable action taken.

## Returns

The setjmp function returns zero to indicate a direct return from the function call. When a call of longjmp causes the return from setjmp, the value returned is the integer passed in the val parameter when longjmp was called, or one if the val parameter was zero.

## Related functions

longjmp

**Example**

```
#include <setjmp.h>

jmp_buf error;

main()
{ switch (setjmp(error))
    { case 0: /* No error yet */
              calculate();
              puts("Program completed successfully");
              exit (0);
      case 1: puts("Unable to open input file");
              break;
      case 2: puts("Invalid input");
              break;
      /* Etc ..... */
    }
  exit(3);   /* indicate an error */
}

void calculate()
{ FILE *input = fopen("input.dat","r");
  if (input == NULL)
    longjmp (error, 1);
  ...
  if (invalid)
    longjmp (error, 2);
  ...
  /* etc */
}
```

# setlocale

The `setlocale` function is used to change or find the current locale.

## Definition

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

## Purpose

This function enables the current locale to changed. The `locale` string is used to define the new locale. In Prospero C, this may be either "C" or "", both meaning the standard C locale. Passing `NULL` for this parameter means that the locale is not changed, but that the current locale can be found. On startup all parts of the locale use the standard C locale.

The parameter `category` is used to define what parts of the locale are to be altered. The following macros (defined in `locale.h`) may be used:-

| Category | Effect on Locale |
|---|---|
| `LC_ALL` | affects all parts of the locale. |
| `LC_COLLATE` | affects the behavior of the `strcoll` function. |
| `LC_CTYPE` | affects the behavior of the character handling functions. |
| `LC_NUMERIC` | affects the decimal point character in formatted input/output functions, and the string conversion functions. |
| `LC_TIME` | affects the behavior of the `strftime` function. |

Note that as only the C locale is supported, no observable effect on functions is produced by `setlocale`.

## Returns

The `setlocale` function returns the string for the specified `category` if `locale` was a string pointer. If `locale` was a `NULL` pointer, then a pointer to the string for the specified `category` under the current locale is returned. If category was `LC_ALL` then a `NULL` pointer is returned, unless the most recent call to set the locale also used `LC_ALL` for the `category`. The string returned may be overwritten by subsequent calls to `setlocale`.

## Related functions

`printf, scanf, strftime`

**Example**

```
#include <locale.h>

/* Restore standard locale */
setlocale (LC_ALL, "C");
```

# setvbuf

The `setvbuf` function allows the buffering of a stream to be controlled.

## Definition

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buf,
             int mode, size_t size);
```

## Purpose

This function is used after opening a stream, but before any input or output has been made, to control how input and output to the file specified by `stream` will be buffered. The `mode` argument should be one of the following values (declared as macros in `stdio.h`) :-

| Mode | Meaning |
|---|---|
| `_IOFBF` | Input/output is fully buffered |
| `_IOLBF` | Input/output is line buffered (the buffer is flushed when full or when a newline character is output, or when input is requested). |
| `_IONBF` | Input/output is completely unbuffered. |

The parameter size specifies the size of the buffer (in bytes). If the parameter buf is not `NULL`, the array to which it points will be used as the buffer – note that this should be aligned on an even word boundary, and must stay in existence for as long as the stream is open. If `buf` is `NULL`, a buffer of size `size` bytes will be automatically allocated, and released when the file is closed.

## Returns

The `setvbuf` function returns zero if successful. If the request is invalid or cannot be met, a non-zero result is returned, and `errno` may be set.

## Related functions

`setbuf, fopen`

**Example**

```
#include <stdio.h>

long int bufsize = 16384;

main()

{  FILE *stream = fopen ("text", "w+");

   void *buffer = malloc (bufsize);

   setvbuf (stream, buffer, _IOLBF, bufsize);

   /* file output is now buffered */

   free ( buffer);
}
```

# signal

The `signal` function controls how signals raised by the raise function are processed.

## Definition

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

## Purpose

This function causes the signal specified by `sig` to be handled in one of three ways. If the parameter `func` is equal to `SIG_IGN`, the signal will be ignored. If func is `SIG_DFL` (both these are macros defined in `signal.h`), the signal will be handled in the default manner. Otherwise, `func` should be a pointer to a function taking a single `int` parameter and returning `void`. In this case, when a signal occurs, first the handling of that signal is restored to its default by the equivalent of `signal(sig, SIG_DFL)`, then the function pointed to by `func` will be called with its parameter defining the signal.

The permitted values of `sig`, and their meanings, are as follows:-

| Signal | Meaning |
|---|---|
| `SIGABRT` | Abnormal program termination (e.g., the abort function is called) |
| `SIGFPE` | Erroneous arithmetic operation (e.g. overflow or zero divide). |
| `SIGILL` | Detection of an illegal function image. |
| `SIGINT` | Receipt of an interactive attention signal |
| `SIGSEGV` | Invalid memory access |
| `SIGTERM` | Receipt of a termination request. |

The above macros, which are defined in `signal.h`, originate in UNIX systems. Prospero C does not generate any of the above signals, except when the `raise` function is called (either explicitly or by calling `abort`). At program startup, all signals are handled in the default manner, which is `SIG_DFL`.

## Returns

The `signal` function returns the value of `func` corresponding to the previously installed handler for the given signal. If the request is invalid or cannot be met, the value of the macro `SIG_ERR` (defined in `signal.h`) is returned, and `errno` will be set.

### Related functions

raise

### Example

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

void notify (int ignored)
{   printf ("Program about to abort, press RETURN");
    scanf ("");
}

signal (SIGABRT, &notify);
```

# sin

The `sin` function computes the sine of a value.

## Definition

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

## Purpose

This function computes the sine of `x` (in radians). If the value of `x` is large, the result may lose some or all significance (returning zero). This will not cause `errno` to be set.

## Returns

The `sin` function returns the sine, in the range –1 to 1.

## Related functions

`cos, tan`

## Example

```
#include <math.h>

printf ( "sin pi = %f\n", sin (3.141592654) );
```

# sinh

The `sinh` function computes the hyperbolic sine of a value.

## Definition

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

## Purpose

This function computes the hyperbolic sine of `x`. If the value of `x` is too large for the result to be represented, a range error occurs. In this case, `errno` will be set to the vale `ERANGE`, and the value `±HUGE_VAL` will be returned, according to the sign of the true result.

## Returns

The `sinh` function returns the hyperbolic sine.

## Related functions

`cosh, tanh`

## Example

```
#include <math.h>

printf ("sinh 1 = %f\n", sinh (1.0) );
```

# sleep

The `sleep` function causes execution to be suspended for a time. It is not part of the draft ANSI standard.

### Definition

```
#include <process.h>
void sleep (unsigned int seconds);
```

### Purpose

This function causes execution to be suspended until the number of seconds specified have elapsed. It is preferable to a delay loop as the delay period will be constant on different machines.

### Returns

There is no return value.

### Related functions

`clock`

### Example

```
#include <process.h>

/* Wait for ten seconds */
sleep(10);
```

# spawn...

The spawnl and related functions transfer execution to another program. They are not part of the draft ANSI standard.

## Definition

```
#include <process.h>
int spawnl  (int mode, const char *path, ...);
int spawnle (int mode, const char *path, ...);
int spawnlp (int mode, const char *path, ...);
int spawnlpe(int mode, const char *path, ...);
int spawnv  (int mode, const char *path,
             const char **args);
int spawnve (int mode, const char *path,
             const char **args, const char **envp);
int spawnvp (int mode, const char *path,
             const char **args);
int spawnvpe(int mode, const char *path,
             const char **args, const char **envp);
```

## Purpose

These functions load and execute the program whose name is given by path as a child process. When the child process terminates, control returns to the calling program. The functions differ in the way that information is made available to the child process. This information falls into two categories, the environment, and the program arguments.

The program arguments are the strings which are passed to the child process's main function in the argv array. By convention, the first of these should be the name of the program. Under GEMDOS this is not available to the executed program, and argv[0] will point to the empty string "". In the functions spawnl, spawnle, spawnlp and spawnlpe, pointers to the program arguments are passed as the second and subsequent parameters, with a NULL pointer being used to indicate the end of the list. In the functions spawnv, spawnve, spawnvp and spawnvpe, the pointers to the program arguments are stored in an array (with a NULL pointer marking the end of the array), and a pointer to this array is passed in the argument args.

The environment is a collection of strings of the form VARIABLE=VALUE, which may be read using the getenv function. Normally a program's environment is the same as that of the program which executed it. However, the functions spawnle, spawnlpe, spawnve and spawnvpe allow a new environment to be specified. These are specified by storing the pointers to

strings of the correct form in an array (with a NULL pointer marking the end of the array), and a pointer to this array is passed in the final argument to the function.

The final variation in these functions concerns how the program to be executed is located. All the functions will look for a file of the given name – if no extension is specified they will attempt to locate it with (in order) no extension, then the extensions .PRG, .TTP, .TOS. If the program has still not been located, and the filename does not specify a drive or pathname, then the spawnlp, spawnlpe, spawnvp and spawnvpe functions will also search (using the same extensions as above) in all directories given by the PATH environment variable.

The mode parameter specifies where the child program should be loaded. If the value is P_WAIT, the child is loaded into memory above the parent, and execution of the parent resumes when the child terminates. If the value is P_OVERLAY (these macros are defined in process.h), the child overwrites the parent process. Only the P_WAIT mode is currently supported by Prospero C under GEMDOS, and any other value of mode will cause an error.

### Returns

If the specified file is successfully found and executed, the return value will be the return code specified by the program when it terminated. For C coded child programs, this is the value returned by its main function or specified as the parameter to exit, or the value EXIT_FAILURE if the program terminated by calling abort. If an error occurs, –1 is returned, and errno will be set.

### Related functions

exit, abort

### Example

The following all pass the arguments "arg1" and "arg2" to the program "child".

```
#include <process.h>

char *environ[] = { "LIB=C:\" , "INCLUDE=C:\H", NULL };
char *args[] = { "child", "arg1", "arg2", NULL};

spawnl  (P_WAIT, "child","child","arg1","arg2", NULL);
spawnle (P_WAIT, "child","child","arg1","arg2", NULL,
         environ);
spawnlp (P_WAIT, "child","child","arg1","arg2", NULL);
spawnlpe(P_WAIT, "child","child","arg1","arg2", NULL,
         environ);
spawnv  (P_WAIT, "child", args);
spawnve (P_WAIT, "child", args, environ);
spawnvp (P_WAIT, "child", args);
spawnvpe(P_WAIT, "child", args, environ);
```

# sprintf

The `sprintf` function writes formatted output to a string.

## Definition

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

## Purpose

This function writes a string of characters controlled by the string pointed to by `format` into the character array pointed to by `s`, followed by a null character. See the description of the function `printf` for details on the `format` string. The function takes a variable number of arguments – the type and number of the arguments after the `format` argument is determined by the layout of the string pointed to by `format`.

## Returns

The `sprintf` function returns the number of characters written to the string, not including the terminating null character.

## Related functions

`fprintf, vfprintf, vprintf, vsprintf`

## Example

```
#include <stdio.h>

char buffer[100];

char *day = "Tuesday";

spintf (buffer, "Today is %s", day);
```

# sqrt

The `sqrt` function computes the square root of a non-negative number.

## Definition

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

## Purpose

This function calculates the non-negative square root of its argument `x`.

## Returns

The `sqrt` function returns the square root. If `x` is negative, a domain error occurs – the value 0.0 is returned, and `errno` will be set to `EDOM`.

## Related functions

`pow, log`

## Example

```
#include <math.h>

double adj = 3.0;
double opp = 4.0;

double hyp = sqrt ( (adj * adj) + (opp * opp) );
```

# srand

The `srand` function sets the seed for the random number generator.

## Definition

```
#include <stdlib.h>
void srand (int seed);
```

## Purpose

This function sets the starting point of the pseudo-random sequence generated by the rand function to be `seed` – at program startup the seed will be zero, and calling `srand` with `seed` zero will cause the same sequence of numbers to be generated as would be if `rand` was called at the start of the program.

## Returns

There is no return value.

## Related functions

`rand`

## Example

```
#include <stdlib.h>
int heads = 0, tails = 0;
srand(5);
if (rand() > 0x3fff)
  heads++;
else
  tails++;
srand(5);
if (rand() > 0x3fff)
  heads--;
else
  tails--;
/* Both heads and tails will now be zero */
```

# sscanf

The `scanf` function reads formatted input from a string.

### Definition

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

### Purpose

This function reads formatted input from the string `s`, under the control of the format string `format`. Additional pointer arguments may be passed, and conversion specifiers in the format string (described in detail in the function `scanf`) will cause input items to be to be converted and their values assigned to the objects pointed to by these arguments.

The `sscanf` function is equivalent to `scanf`, except that the input is obtained from the string `s` rather than from standard input, and encountering the end of the string is equivalent to encountering end-of-file in `scanf`.

### Returns

The `scanf` function returns the number of input items assigned, which may be less than the number provided for or zero if a match failure occurs. If an input failure occurs before any conversion, `EOF` is returned.

### Related functions

`fscanf, scanf, strtod, strtol, strtoul`

### Example

```
#include <stdio.h>

char data[10];

sscanf ("Today is Thursday", "Today is %s", data);
printf ("%s\n", data);
```

will print out

```
Thursday
```

# strcat

The `strcat` function concatenates two null terminated strings.

## Definition

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

## Purpose

This function appends a copy of the null terminated string pointed to by `s2` to the end of the null terminated string pointed to by `s1`, with the first character of `s2` replacing the terminating null of `s1`. The two strings should not overlap.

## Returns

The `strcat` function returns a pointer to the (modified) string `s1`.

## Related functions

`strncat, strcpy`

## Example

```
#include <string.h>

char a[20] = "first";
char * b = "second";

strcat (a, b);

printf("%s\n", a);
```

will produce as output

```
firstsecond
```

# strchr

The `strchr` function searches for a character in a null terminated string.

### Definition

```
#include <string.h>
char *strchr (const char *s, int c);
```

### Purpose

This function searches for the first occurrence of the character `c` (converted to `char`) in the null terminated string pointed to by `s`. Note that the terminating null character is included in the search.

### Returns

The `strchr` function returns a pointer to the first instance of the character `c`, or `NULL` if no instance is found.

### Related functions

`memchr, strcspn, strpbrk, strrchr, strspn, strstr`

### Example

```
#include <string.h>

char *ptr = strchr ("abcdefghijklmnop", 'f');

printf ("after f comes %c\n", *(ptr + 1) );
```

produces as output

```
after f comes g
```

# strcmp

The `strcmp` function compares two null terminated strings.

## Definition

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

## Purpose

This function compares the null terminated strings pointed to by `s1` and `s2`. The characters are always treated as unsigned, regardless of the compilation options.

## Returns

The `strcmp` function returns an integer less than, equal to, or greater than zero, according to whether the string pointed to by `s1` is less than, equal to, or greater than that pointed to by `s2`.

## Related functions

`stricmp, strncmp, strnicmp, memcmp, memicmp`

## Example

```
#include <string.h>

struct node {char name[20];
             struct node *left;
             struct node *right; };

struct node *tree_search(struct node *n, char *target)
{ int diff;

  if (n == NULL) return NULL;

  diff = strcmp (n->name, target);
  if (diff == 0) return n;
  return tree_search(diff < 0 ? n->left : n->right,
                     target);
}
```

# strcoll

The `strcoll` function transforms a string for comparison according to the current locale.

## Definition

```
#include <string.h>
size_t strcoll (char *to, size_t maxsize,
                const char *s2);
```

## Purpose

This function transforms a string so that if two transformed strings are compared using `memcmp`, `strcmp` or other library compare function, then the result will be appropriate to the current locale. No change occurs if the current locale is the standard locale. No more than `maxsize` characters will be placed in the transformed string `to` (including the terminating null character). However, the length of the resulting string will be at most twice the length of the original string (plus the terminating null character).

## Returns

The `strcoll` function returns the length of the resulting string (not including the terminating null character). If the resulting string is larger than `maxsize` characters then zero is returned and the contents of `to` will be indeterminate.

## Related functions

`setlocale, stricmp, strncmp, strnicmp, memcmp, memicmp`

**Example**

```
#include <string.h>

char string1[6], string2[6];

/* setlocale here */

strcpy(string1, "Hello");
strcoll(string2, 6, "Hello");

if (strcmp(string1,string2))
   puts("The transformed string is different");
else
   puts("The transformed string has not changed");
```

# strcpy

The `strcpy` function copies one null-terminated string to another.

## Definition

```
#include <string.h>
char *strcpy (char *dst, const char * src);
```

## Purpose

This function copies characters from the string pointed to by `src` to the array pointed to by `dst`, up to and including the terminating null character. The destination object must be large enough to hold this string (i.e. at least `strlen(src) + 1` characters), or unpredictable results will occur. The two strings should not overlap.

## Returns

The `strcpy` function returns the address of the destination string, `dst`.

## Related functions

`strncpy, strdup, memcpy, memmove`

## Example

The `strcat` function could have been coded as follows:

```
#include <string.h>

/* A simple way of defining strcat */
char *strcat (char *dst, const char *src)

{ char *a = dst;
  while (*a)
    a++;
  /* a now points to dst's terminating null */
  strcpy (a, src);
  return dst;
}
```

# strcspn

The `strcspn` function returns the length of the initial segment of a null terminated string which consists entirely of characters not occurring in another string.

## Definition

```
#include <string.h>
size_t strcspn (const char *str, const char *template);
```

## Purpose

This function locates the first character in `str` which also occurs in `template`, and returns the number of characters which precede it. If `template` is empty, or contains no characters which occur in `str`, the result will be equal to `strlen(str)`.

## Returns

The `strcspn` function returns the length of the initial segment of `str` which contains no characters which also occur in `template`. This is equivalent to the index within `str` of the first character which is also in `template` (if the null-terminating character is considered part of `template`).

## Related functions

`strspn, strpbrk`

## Example

```
#include <string.h>

char *filename;

if (strcspn (filename, ".") > 8)
   error("Filename too long");
```

# strdup

The `strdup` function duplicates a null terminated string. This function is not part of the draft ANSI standard.

## Definition

```
#include <string.h>
char *strdup (const char *src);
```

## Purpose

This function duplicates the string `src`. A block of memory of size `strlen(src) + 1` is allocated (using `malloc`), and the string pointed to by `src` is copied into it, up to and including the terminating null character. The memory allocated should be released (using `free`) when no longer required.

## Returns

The `strdup` function returns a pointer to the new copy of the string. If no memory can be allocated, it returns `NULL`.

## Related functions

`strcpy`

## Example

```
#include <string.h>

/* Output a string backwards */

void reverse_puts(const char *str)
{ char * localcopy;

  localcopy = strdup(str);
  if (localcopy)
     puts(strrev(localcopy));
  free(localcopy);
}
```

# strerror

The `strerror` function returns a string describing a given error number.

## Definition

```
#include <string.h>
char *strerror (int errnum);
```

## Purpose

This function maps the error number `errnum` to an error message string. Typically this will be used for generating error messages when a library routine fails, and the value of the macro `errno` will be passed as the parameter `errnum`. Note that this function (as defined in the ANSI standard) is not identical to the `strerror` function available on some UNIX systems.

## Returns

The `strerror` function returns a pointer to the relevant error message string. If the value of `errnum` is not one of the error codes described in the header file `errno.h`, this will be an empty string, otherwise it will contain the same text as the comment in `errno.h`. The string should not be modified by the program.

## Related functions

`perror`

## Example

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

/* Open a file carefully */
FILE * careful_open(const char *name, const char *mode)
{  FILE * thefile;
   errno = 0;
   thefile = fopen(name, mode);
   if (thefile == NULL)
      fprintf(stderr, "Error opening file %s : %s\n",
              name, strerror(errno));
   return thefile;
}
```

# strftime

The `strftime` function constructs a string describing a given date and time in a given format.

## Definition

```
#include <time.h>
size_t strftime (char * s, size_t maxsize,
         const char *format, const struct tm *timeptr);
```

## Purpose

This function places up to `maxsize` characters in the array pointed to by `s`, under the control of the format string pointed to by `format`. Any character in the format string not preceded by a % character is copied into the array unchanged, including the terminating null. Directives in the format string, which consist of the % character followed by a letter, cause characters to be placed in the array determined by the directive, the current locale, and the date and time specified by the structure pointed to by the parameter `timeptr` :-

| Directive | Result |
|---|---|
| %a | locale's abbreviated weekday name |
| %A | locale's full weekday name |
| %b | locale's abbreviated month name |
| %B | locale's full month name |
| %c | locale's appropriate date and time representation |
| %d | day of the month (01 - 31) |
| %H | hour using 24 hour clock (00 - 23) |
| %I | hour using 12 hour clock (01 - 12) |
| %j | day of the year (001 - 366) |
| %m | month of the year (01 - 12) |
| %M | minute (00 - 59) |
| %p | locale's equivalent of AM or PM |
| %S | second (00 - 59) |
| %U | week number, with Sunday as the first day of the week (00 - 53) |
| %w | weekday as a number, where 0 means Sunday (0 - 6) |
| %W | week number, with Monday as the first day of the week (00 - 53) |
| %x | locale's appropriate date representation |
| %X | locale's appropriate time representation |
| %y | year without century (00 - 99) |
| %Y | year with century |
| %Z | timezone name, or no characters if no timezone exists. This is determined by the value of the environment variable TZ. |
| %% | a single % character |

### Returns

If the time could be represented in the given format in no more than `maxsize` characters (including the null), the `strftime` function returns the number of characters placed in the array `s` (excluding the terminating null). Otherwise, it returns zero, and the contents of the array will be indeterminate.

### Related functions

`setlocale, asctime`

### Example

```
#include <time.h>

main()
{   char timestring[40];
    time_t now = time(NULL);


    if (strftime (timestring, 40,
                  "It is now %I:%M %P",
                  localtime(now) )
       puts(timestring);
}
```

# stricmp

The `stricmp` function compares two null terminated strings, ignoring case differences. This function is not part of the draft ANSI standard.

## Definition

```
#include <string.h>
int stricmp (const char *s1, const char *s2);
```

## Purpose

This function compares the null terminated strings pointed to by `s1` and `s2`, treating upper and lower case letters as equivalent. The characters are always treated as unsigned, regardless of the compilation options.

## Returns

The `stricmp` function returns an integer less than, equal to, or greater than zero, according to whether the string pointed to by `s1` is less than, equal to, or greater than that pointed to by `s2`, with all letters regarded as if they were upper case in both strings.

## Related functions

`setlocale, strcmp, strncmp, strnicmp, memcmp, memicmp`

## Example

```
#include <string.h>

int f_option = 0;

main(int argc, char *argv[])
{ int cnt;
  for (cnt = 0; cnt < argc; cnt ++)
    if (stricmp (argv[cnt], "-f"))
      puts("Unrecognized option");
    else
      f_option++;

  ...
}
```

# strlen

The `strlen` function computes the length of a null terminated string.

## Definition

```
#include <string.h>
size_t strlen (const char *s);
```

## Purpose

This function computes the length of the null terminated string pointed to by `s`. The value does not include the terminating null itself.

## Returns

The `strlen` function returns the number of characters before the null character which terminates the string `s`.

## Example

```
#include <string.h>

  char buffer[20];

  if (strlen (s) <= 19)
    strcpy(buffer, s);
  else
  {  puts("Name too long");
     exit (1);
  }
```

# strlwr

The `strlwr` function converts a string to lower case. This function is not part of the draft ANSI standard.

### Definition

```
#include <string.h>
char *strlwr (char *s);
```

### Purpose

This function converts all upper case characters in the string pointed to by `s` to lower case. All other characters, up to and including the terminating null character, remain unaltered.

### Returns

The `strlwr` function returns the address of the original string, `s`.

### Related functions

`strupr, tolower, toupper`

### Example

```
#include <string.h>

char *s = "Hello world!";

puts(strlwr(s));
```

produces the output

```
hello world!
```

# strncat

The `strncat` function appends characters from one null-terminated string to another.

## Definition

```
#include <string.h>
char *strncat (char *dst, const char *src, size_t n);
```

## Purpose

This function appends characters from the string pointed to by `src` to the end of the string pointed to by `dst`, until either the terminating null character is reached or `n` characters have been copied. A null character is then appended to the result. The first character of `src` overwrites the null at the end of `dst`. The destination object must be large enough to hold the resulting string (i.e. it must have at least `strlen(dst) + n + 1` characters), or unpredictable results will occur. The two strings should not overlap.

## Returns

The `strncat` function returns the address of the destination string, `dst`.

## Related functions

`strcat`

## Example

```
#include <string.h>

main()
{   char salutation[20], *surname;

    ...
    strcpy(salutation, "Dear Mr ");
    strncat(salutation, surname, 20 - 8 - 1);
    ...
}
```

# strncmp

The `strncmp` function compares the initial portions of two null-terminated strings.

## Definition

```
#include <string.h>
int strncmp (const char *s1, const char *s2, size_t n);
```

## Purpose

This function compares not more than `n` characters from the null-terminated strings pointed to by `s1` and `s2`. The characters are always treated as unsigned, regardless of the compilation options.

## Returns

The `strncmp` function returns an integer less than, equal to, or greater than zero, according to whether the string pointed to by `s1` is less than, equal to, or greater than that pointed to by `s2`, ignoring any characters after the `n`'th but before the end of either string, if both strings are longer than `n`.

## Related functions

`strcmp, stricmp, strnicmp, memcmp, memicmp`

## Example

```
#include <string.h>

/* Check if sentence starts with a particular word */

char *sentence;
int furthermores = 0;

if (strncmp(sentence, "Furthermore ", 12) == 0)
   furthermores++;
```

# strncpy

The `strcpy` function copies the initial portion of one null-terminated string to another.

## Definition

```
#include <string.h>
char *strncpy (char *dst, const char* src, size_t n);
```

## Purpose

This function copies `n` characters from the string pointed to by `src` to the array pointed to by `dst`. If `src` contains less than `n` characters, null characters will be appended to `dst` until `n` characters in all have been written. If `src` contains more then `n` characters, the string pointed to by `dst` will not be null terminated. The destination object must be large enough to hold this string (i.e. at least `n` characters), or unpredictable results will occur. The two strings should not overlap.

## Returns

The `strncpy` function returns the address of the destination string.

## Related functions

`strcpy, strdup, memcpy, memmove`

## Example

```
#include <string.h>

   char buffer[10], *str;

   /* Copy first 9 chars of a string */
   strncpy(buffer, str, 9);

   /* Note that this may not be null terminated, so
      do so now */
   buffer[9] = 0;
```

# strnicmp

The strncmp function compares the initial portions of two null-terminated strings, ignoring case differences. This function is not part of the draft ANSI standard.

## Definition

```
#include <string.h>
int strnicmp(const char *s1, const char *s2, size_t n);
```

## Purpose

This function compares not more than n characters from the null-terminated strings pointed to by s1 and s2, treating upper and lower case letters as identical. The characters are always treated as unsigned, regardless of the compilation options.

## Returns

The strnicmp function returns an integer less than, equal to, or greater than zero, according to whether the string pointed to by s1 is less than, equal to, or greater than that pointed to by s2, ignoring any characters after the n'th but before the end of either string, if both strings are longer than n. All letters in either string are treated as if they were upper case.

## Related functions

strcmp, stricmp, strncmp, memcmp, memicmp

## Example

```
#include <string.h>

char *clause;
int howevers = 0;

/* Check if clause starts with a particular word */

if (strnicmp(clause, "however ", 12) == 0)
   howevers++;
```

# strnset

The `strnset` function fills the initial portion of a string with a character. It is not part of the draft ANSI standard.

### Definition

```
#include <string.h>
char *strnset (char *s, int c, size_t n);
```

### Purpose

This function sets the characters in the string `s` to the value `c` (converted to an `unsigned char`), until `n` characters have been written or the null-terminating character is reached.

### Returns

The `strnset` function a pointer to the destination string, `s`.

### Related functions

`memset, strset`

### Example

```
#include <string.h>

char s[20];

strcpy (s, "Hello there\n");
puts (strnset (s, '*', 5));
```

prints out the string

```
***** there
```

# strpbrk

The `strpbrk` function locates the first character from a given set in a string.

### Definition

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

### Purpose

This function locates the first instance in the string pointed to by `s1` of any character from the string pointed to by `s2`. The terminating null characters are not included in the search.

### Returns

The `strpbrk` function returns a pointer to the first such character. If no characters occur in both strings, `NULL` is returned.

### Related functions

`strstr, strchr, strspn, strtok`

### Example

```
#include <string.h>

main()
{  char *s;
   puts (strpbrk("What a surprise!\n", "aeiouAEIOU") );
}
```

produces the output

```
at a surprise!
```

# strrchr

The strrchr function searches for a character in a null terminated string.

## Definition

```
#include <string.h>
char *strrchr (const char *s, int c);
```

## Purpose

This function searches for the last occurrence of the character c (converted to char) in the null terminated string pointed to by s. Note that the terminating null is included in the search.

## Returns

The strrchr function returns a pointer to the last (rightmost) instance of the character c, or NULL if no instance is found.

## Related functions

memchr, strcspn, strpbrk, strchr, strspn, strstr

## Example

```
#include <string.h>

char *pathname, *filename;

/* Print out the filename portion of a path name */

if ( (filename = strrchr(pathname, '\\')) != NULL)
   puts(filename);
else
   puts(pathname);
```

# strrev

The `strrev` function reverses a null terminated string.

### Definition

```
#include <string.h>
char *strrev (char *s);
```

### Purpose

This function reverses the order of the characters in the null-terminated string pointed to by `s`. Note that as the contents of the array pointed to by `s` are modified, a string constant should not be passed.

### Returns

The `strrev` function returns a pointer to the destination string, `s`.

### Related functions

`strcpy`

### Example

```
#include <string.h>

char *s;
strcpy (s, "Have a nice day!");
strrev (s);
puts (s);
```

produces the output

```
!yad ecin a evaH
```

# strset

The `strset` function fills a string with a character. It is not part of the draft ANSI standard.

### Definition

```
#include <string.h>
char *strset (char *s, int c);
```

### Purpose

This function sets all the characters in the string pointed to by `s` which precede the terminating null to the value `c` (converted to an `unsigned char`).

### Returns

The `strset` function returns a pointer to the destination string, `s`.

### Related functions

`memset, strnset`

### Example

```
#include <string.h>

char s[12];

strcpy (s, "Hello there");
puts (strset (s, '*'));
```

prints out the string

```
**********
```

# strspn

The `strspn` function measures the span of characters from a given set in a null terminated string.

## Definition

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

## Purpose

This function computes the number of characters at the start of the string pointed to by `s1` which are also in the string pointed to by `s2`. This is also the index of the first character in `s1` which is not also in `s2`. The terminating null characters are not included in the scans.

## Returns

The `strspn` function returns the number of matching characters.

## Related functions

`strcspn, strpbrk, strchr, strstr`

## Example

```
#include <string.h>

char *name = "Prospero Software";
int len;

/* Calculate length of first name */

len = strspn(name, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                   "abcdefghijklmnopqrstuvwxyz");

/* len will be 8 here */
```

# strstr

The `strstr` function locates one null terminated string in another.

### Definition

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

### Purpose

This function locates the first occurrence of the string pointed to by `s2` in the string pointed to by `s1`. The terminating null character of `s2` is not included in the search.

### Returns

The `strstr` function returns a pointer to the start of the first occurrence found. If no occurrence is located, `NULL` is returned. If `s2` is empty, a pointer to `s1` is returned.

### Related functions

`strchr`

### Example

```
#include <string.h>

char *s = "Happy birthday to you!";

puts (strstr(s, "day"));
```

produces the output

```
day to you!
```

# strtod

The strtod function converts a string to a floating point number.

## Definition

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
```

## Purpose

This function attempts to convert the string pointed to by nptr into a floating point number. First, leading white space characters (as defined by isspace) are skipped. Next, a series of characters are read to form the subject sequence. This should start with an optional plus or minus sign, followed by a sequence of one or more digits, optionally including a decimal-point character, followed by an optional exponent, which consists ot the letter "E" or "e", a plus or minus sign, and a sequence of one or more digits. As soon as a character is encountered which does not agree with the above sequence, scanning stops.

If a subject sequence matching the above is read, it is converted to a double precision value, and a pointer to the first character after the subject sequence is assigned to the object pointed to by endptr, unless endptr is NULL. If the subject sequence does not match the above description, the value of nptr is copied to the object pointed to by endptr.

## Returns

The strtod function returns the converted value. If no matching sequence is found, zero is returned, and *endptr will be equal to nptr (unless endptr is NULL). If the converted value would overflow, plus or minus HUGE_VAL is returned, and errno will be set to ERANGE. If the converted value would underflow, zero is returned, and errno will be set to ERANGE.

## Related functions

atof, atoi, scanf, sscanf , strtol, strtoul

**Example**

```
#include <stdlib.h>

char *s;
double d;

d = strtod("-1.23E20.5", &s);

printf("d is %g, s is '%s' \n", d, s);
```

prints out the string

```
d is -1.23e20, s is '.5'
```

# strtok

The `strtok` function breaks a string into tokens.

## Definition

```
#include <string.h>
char *strtok (char *s1, const char *s2);
```

## Purpose

This function splits the string pointed to by `s1` into a sequence of tokens, delimited by characters in the string `s2`. This is done by making a sequence of calls to the `strtok` function as follows.

The first call of a sequence takes a pointer to a subject string to be tokenized in the parameter `s1`, and a pointer to a string containing delimiter characters in `s2`. The function searches through the string pointed to by `s1` until a character not in `s2` is found – this is the start of the first token. Subsequent calls in the sequence pass a `NULL` pointer as `s1`, and the value of the start of the next token stored by the last call in the sequence is used.

The function then searches for the next character in the subject which is also in the set of delimiters in `s2`. If there is none, the current token extends to the end of the subject string, and subsequent calls in the same sequence will return `NULL`. Otherwise, the delimiting character is overwritten with a null character to terminate the current token, and a pointer to the character following it is stored for use as the first character of the next token.

The delimiter string pointed to by `s2` need not be the same for different calls in the same sequence.

Note that the subject string will be modified, as delimiter characters are overwritten by nulls. A string constant should not therefore be used as the first parameter.

## Returns

The `strtok` function returns a pointer to the start of the token, or `NULL` if there are no tokens.

### Related functions

strcspn, strspn

### Example

```
#include <string.h>

char * token, subject;

strcpy (s, "abc,def.4");

token = strtok(s, ",");        /* token is "abc" */
token = strtok(NULL, ",#f")    /* token is "de"  */
token = strtok(NULL, ".")      /* token is "4"   */
token = strtok(NULL, "#")      /* token is NULL  */
```

# strtol

The strtol function converts a string to an integer.

## Definition

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr,
                 int base);
```

## Purpose

This function attempts to convert the string pointed to by nptr into an integer using the radix specified by base. First, leading white space characters (as defined by isspace) are skipped. Next, a series of characters are read to form the subject sequence.

The expected subject sequence depends upon the value of base as follows (all subject sequences can be preceded by an optional plus or minus sign). If base is zero, the subject sequence should be in the same form as an octal, decimal or hexadecimal C constant (without a suffix), and the base to be used for the conversion is determined automatically from the form of the constant. Otherwise, base should be in the range 2 to 36, and the subject sequence consists of digits and letters representing values less than the given base, where the letters "A" to "Z" (in either upper or lower case) represent the values 10 to 35. If base is equal to 16, the subject sequence may start with an optional "0x" or "0X" sequence.

If a subject sequence matching the above is read, it is converted to a long int value, and a pointer to the first character after the subject sequence is assigned to the object pointed to by endptr, unless endptr is NULL. If the subject sequence does not match the above description, the value of nptr is copied to the object pointed to by endptr.

## Returns

The strtol function returns the converted value. If no matching sequence is found, zero is returned, and *endptr will be equal to nptr (unless endptr is NULL). If the converted value would overflow, LONG_MAX or LONG_MIN is returned, depending on the sign of the result, and errno will be set to ERANGE.

### Related functions

atof, atoi, scanf, sscanf, strtod, strtoul

### Example

```
#include <stdlib.h>

char *s;
long int l;

l = strtol("1011 Fred", &s, 2);

printf("l is %ld, s is '%s'\n", l, s);
```

prints out the string

```
l is 11, s is ' Fred'
```

# strtoul

The strtoul function converts a string to an unsigned long integer.

## Definition

```
#include <stdlib.h>
unsigned long strtol (const char *nptr, char **endptr,
                      int base);
```

## Purpose

This function attempts to convert the string pointed to by nptr into an integer using the radix specified by base. Apart from the return value, it is equivalent to strtol, but may be useful for converting values which would overflow the range of signed long integers. Note that the subject sequence may still be preceded by a minus sign, which will cause the value to be negated before being returned.

## Returns

The strtoul function returns the converted value. If no matching sequence is found, zero is returned, and *endptr will be equal to nptr (unless endptr is NULL). If the converted value would overflow, ULONG_MAX is returned, and errno will be set to ERANGE.

## Related functions

atof, atoi, scanf, sscanf, strtod, strtol

## Example

```
#include <stdlib.h>

char * s;
unsigned long int l;

l = strtoul("-1 Fred", &s, 10);

printf("l is %lx, s is '%s'\n", l, s);
```

prints out the string

```
l is 0xffffffff, s is ' Fred'
```

# strupr

The `strupr` function converts a string to upper case. This function is not part of the draft ANSI standard.

### Definition

```
#include <string.h>
char *strupr (char *s);
```

### Purpose

This function converts all lower case characters in the string pointed to by `s` to upper case. All other characters, up to and including the terminating null character, remain unaltered.

Note that the string pointed to by `s` is modified, and therefore a string literal should not be passed.

### Returns

The `strupr` function returns a pointer to the original string, `s`.

### Related functions

`strlwr, tolower, toupper`

### Example

```
#include <string.h>

main()
{  char s[13];
   strcpy(s, "Hello world!");
   puts (strupr(s));
}
```

produces the output

```
HELLO WORLD!
```

# swab

The swab function swaps bytes pairwise, copying them to a new destination. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
void swab (void *source, void *dest, size_t nmemb);
```

## Purpose

This function copies nmemb bytes from source to dest. Each pair of bytes read is swapped before being written to the destination. It is useful for converting data between two processors: one that stores data low byte first, and the other storing data high byte first. Note that nmemb must be even.

## Returns

The swab function returns no value.

## Related functions

memcpy, memmove

## Example

```
#include <stdlib.h>

int data[100], data2[100];
FILE *stream = fopen ("data", "wb+");

/* convert the file data to store bytes in opposite
   order */

fread (data, sizeof (int), 100, stream);
swab (data, data2, 100 * sizeof (int) );
rewind (stream);
fwrite (data, sizeof (int), 100, stream);
fclose (stream);
```

# system

The `system` function calls a command processor.

## Definition

```
#include <stdlib.h>
int system (const char *string);
```

## Purpose

This function calls a command processor. This is not implemented in Prospero C; the function is included here to provide compatibility with other systems. Passing a `NULL` pointer to the function can be used to determine if a command processor is available.

## Returns

The `system` function always returns zero to indicate no command processor is available.

## Related functions

`bios, gemdos, xbios`

## Example

```
#include <stdlib.h>

if (system (NULL) != 0)
   {  /* Use command processor */
      ...
   }
else
   puts ("No command processor is available");
```

# tan

The tan function computes the tangent of a value.

## Definition

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

## Purpose

This function calculates the tangent of the argument x, specified in radians.

## Returns

The tan function returns the tangent of x (in radians). If the value of x is very large, the result may lose some or all significance – in the latter case, zero will be returned, but no error will be recorded in errno.

Note that it is impossible to cause overflow as the compiler cannot represent $\pi/2$ etc. closely enough for this to occur.

## Related functions

atan, atan2, cos, sin, tanh

## Example

```
#include <math.h>

#define pi 3.1415926535

int i;
double x;

for (i = 0; i < 10; i++)
  {  x = i * pi / 10;
     printf ("tan(%f) is %f\n", x, tan(x));
  }
```

# tanh

The `tanh` function computes the hyperbolic tangent of a value.

## Definition

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

## Purpose

This function calculates the hyperbolic tangent of the argument `x`.

## Returns

The `tanh` function returns the hyperbolic tangent of `x`. There is no error return.

## Related functions

`sinh, cosh, exp`

## Example

```
#include <math.h>

double x;

for (x = 0.0; x < 1.0; x += 0.1)
  printf ("tanh(%f) is %f\n", x, tanh(x));
```

# tell

The `tell` function returns the file position associated with an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long int tell (int handle);
```

## Purpose

This function returns the position of the file pointer (the location within the file at which the next input or output is performed) for the file associated with `handle`. This value may then be used in a subsequent call of `lseek` to restore the file to that position.

## Returns

The `tell` function returns the file position relative to the start of the file. If an error occurs, it returns `-1L`, and `errno` will be set to indicate the error.

## Related functions

`fgetpos, fsetpos, fseek, ftell, lseek`

## Example

```
#include <io.h>

/* function to determine how far we are from the end
   of the file */

long int from_end (int handle)

{   return (long) filelength (handle) - tell (handle);
}
```

# time

The time function reads the current system time.

### Definition

```
#include <math.h>
time_t time (time_t *timer);
```

### Purpose

This function reads the current system time. If timer is not NULL, then the time is also assigned to the object pointed to by timer.

### Returns

The time function returns the date and time, according to the system clock. These are returned in a type of time_t. The functions localtime and gmtime can convert the time in this format to a broken down format, which contains the time components in separate fields, or ctime can be used to convert the time to a string.

### Related functions

asctime, clock, gmtime, localtime, mktime

### Example

```
#include <time.h>

time_t timer = time (NULL);

char *time_buf = ctime (&timer);

printf ("The time is %s\n", time_buf);
```

# tmpfile

The `tmpfile` function creates a temporary binary file.

## Definition

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

## Purpose

This function creates a temporary binary file which will be automatically deleted when it is closed (either explicitly or on normal program termination). The file is opened in the same mode as by using `fopen` with parameter `"wb+"`, and with a filename which does not correspond to any existing file.

## Returns

The `tmpfile` function returns a pointer to the newly created stream information. If unsuccessful, `NULL` is returned.

## Related functions

`fopen, tmpnam`

## Example

```
#include <stdio.h>

main()

{   FILE *stream = tmpfile ();

    /* Use stream as a temporary file */
}
/*  the temporary file is now closed and removed */
```

# tmpnam

The tmpnam function creates a unique file name.

## Definition

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

## Purpose

This function creates a valid file name which is not the same as any existing file. A different filename will be returned each time it is called. The file names generated are of the form "CTEM$nnn.$$$", where nnn represents a number in the range 000 to 999.

## Returns

If the argument s is not NULL, it should point to an array of at least L_tmpnam characters, into which the file name is written. If s is NULL, the filename is written into static store, which may be overwritten by subsequent calls of the tmpnam function. The function returns either s or a pointer to the static store, or NULL if no unique filename could be generated.

## Related functions

tmpfile

## Example

```
#include <stdio.h>

/* Open 5 temporary text files */
FILE *stream[5];
char filename[5,L_tmpnam];
int i;
for( i=0; i<5; i++)
   stream[i] = fopen (tmpnam ( filename[i]), "w+");

  /* use the files */
  ...
fcloseall();     /* close and remove */
for( i=0; i<5; i++)
   remove ( filename[i]);
```

# toascii

The `toascii` function converts a character to the ascii range. It is not part of the draft ANSI standard.

### Definition

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

### Purpose

This function is used to force a character value to the range 0 to 127 where the ascii standard character values apply. It defined as a macro in `ctype.h`, but if `ctype.h` is not `#included`, or if `toascii` is `#undef`'d, a library function will be called.

### Returns

The `toascii` function returns the value of `c`, logically ANDed with `0x7F` to force it to the required range.

### Related functions

`toupper, tolower`

### Example

```
#include <ctype.h>

putc ( toascii ( 193) );
```

will produce an `A`.

# tolower

The `tolower` function converts a character to lower case.

## Definition

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

## Purpose

If the character `c` (converted to an `unsigned char`) is an upper case letter, it is converted to the corresponding lower case letter. Characters other than upper case letters are not converted.

## Returns

The `tolower` function returns the value of `c`, converted to lower case if appropriate.

## Related functions

`toupper, strlwr, strupr`

## Example

```
#include <ctype.h>

putc ( tolower ('A') );
```

will produce an `a`.

# toupper

The `toupper` function converts a character to upper case.

### Definition

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

### Purpose

If the character `c` (converted to an `unsigned char`) is a lower case letter, it is converted to the corresponding upper case letter. Characters other than lower case letters are not converted.

### Returns

The `toupper` function returns the value of `c`, converted to upper case if appropriate.

### Related functions

`tolower, strlwr, strupr`

### Example

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

puts ( "Continue (y/n) ?");

if ( toupper ( getche () ) == 'Y')
    /* continue */
    ...
else
    /* finish */
    ...
```

# tzset

The `tzset` function sets the time zone from the environment. It is not part of the draft ANSI standard.

### Definition

```
#include <time.h>
void tzset (void);
```

### Purpose

This function scans the environment for the environment variable `TZ`. This consists of three letters indicating timezone name and a decimal number indicating how many hours the timezone is earlier than GMT (e.g., `ABC1`); this may optionally be followed by three letters indicating the daylight saving time name and that it is in effect (e.g., `ABC-1BST`). If the environment variable `TZ` is not found, then the timezone will be unknown, and `gmtime` will not be able to find Greenwich Mean Time.

### Returns

No value is returned.

### Related functions

`gmtime, strftime`

### Example

```
#include <time.h>

char buffer[81];
time_t now;

tzset ();
now = time(NULL);
strftime (buffer, 80, "The time is now %H:%M:%S %Z\n",
             localtime(&now) );
```

will print (assuming `TZ=GMT-1BST`)

```
The time now is 14:26:52 BST
```

# ultoa

The `ultoa` function converts an unsigned long integer into an ASCII string. It is not part of the draft ANSI standard.

## Definition

```
#include <stdlib.h>
char *ultoa (unsigned long value, char *string,
             int radix);
```

## Purpose

This function converts the unsigned long integer `value` into ASCII characters in `string`, representing the value of the integer in the base `radix` (in the range 2 to 36). A terminating null character is always appended to the resulting string. The maximum number of characters which can be placed into the array pointed to by `string` is 33 (when `radix` is 2).

## Returns

The `ultoa` function returns the value of `string`.

## Related functions

`itoa, ltoa, sprintf, atoi, atol, strtol, strtoul`

## Example

```
#include <stdlib.h>

char digits[33];
unsigned long int big = 123456789;

ultoa (big, digits, 2);

printf ("%ld in binary is %s\n", big, digits);
```

# ungetc

The ungetc function pushes a character back onto an input stream.

## Definition

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

## Purpose

This function pushes the character c (converted to an unsigned char) back onto the input stream specified by stream, so that the next input operation on that stream will read that character as the first character. The character is not actually written to the file, and will be discarded if a file positioning function is called before the character is read back. A pushed back character must be re-read (or discarded) before another character can be pushed back.

## Returns

The ungetc function returns the character c if successful. If it is called when a character has already been pushed back and not yet read, or if the value of the argument c is EOF, ungetc returns EOF.

## Related functions

getc, getchar, ungetch

## Example

```
#include <stdio.h>

/* function to move file pointer onto next non-space
   char */

void next_char (FILE * stream)

{  int ch;
   while (!feof (stream )
          && isspace (ch = getc ( stream ) ) );
   if (!feof (stream) )
      ungetc (ch, stream);  /* put the char back */
}
```

# ungetch

The `ungetch` function pushes back a character to the console. It is not part of the draft ANSI standard.

## Definition

```
#include <conio.h>
int ungetch (int c);
```

## Purpose

This function causes the character `c` (converted to an `unsigned char`) to be stored, so that the next console input operation using `getch` or `getche` will read back that character. The character is not actually written to the screen. Once a character has been pushed back, it must be re-read before another character is pushed back.

## Returns

The `ungetch` function returns the character `c` if successful. If it is called when a character has already been pushed back and not yet read, or if the value of the argument `c` is `-1`, `ungetch` returns `-1`.

## Related functions

`getch, getche, ungetc`

## Example

```
#include <conio.h>

/* function returns the next character from the
   keyboard as long as it is a letter, else returns -1
   and leaves the character unread */
int get_alpha( void)

{  int ch;
   if ( isalpha ( ch = getche () ) )
       return ch;
   ungetch ( ch);
   return -1;
}
```

# va_arg, va_end, va_start

These macros are used to obtain the arguments from a variable argument list.

## Definition

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

## Purpose

These macros are used to obtain arguments from a variable argument list in a portable manner. To access the parameters of a function accepting a variable number or type of parameters, a pointer of type `va_list` (declared in `stdarg.h`) is initialized to point to the first variable argument by calling the macro `va_start`. The first parameter of this macro is the pointer to be initialized, while the second is the name of the right-most non-variable parameter of the function (the one immediately preceding the elipsis).

Once the pointer is initialized, `va_arg` is used to obtain the parameters. The first call of `va_arg` will yield the first argument, while subsequent calls will give the second, third and so on. The type of the argument to be fetched is given as the second parameter to the macro – this controls the type of the expression to which the macro expands. After the argument is fetched, the pointer `ap` is incremented by the size of *type* to point to the next argument in the list. If the types of the arguments fetched do not match the values passed when the function was called, the results will be undefined.

Finally, any function which calls `va_start` should also call `va_end` with the same first parameter before returning. Under Prospero C, this has no effect, but should be included for portability to other systems. The pointer `ap` should not be used after the call to `va_end`.

There are no library functions corresponding to these macros, and the file `stdarg.h` must be included if they are to be used.

### Returns

The `va_start` and `va_end` macros return no values, but may modify the pointer `ap`. The `va_arg` macro expands to an expression of type *type*, which evaluates to the value of the next parameter in the variable argument list. If the type of this parameter was not *type*, the value returned by this and subsequent calls of `va_arg` will be undefined.

### Example

```
#include <stdarg.h>

int max(int n, ...)
{ va_list args;
  int biggest = INT_MIN;

  va_start (args, n);
  while (n--)
  {  int next = va_arg(args, int);
     if (next > biggest)
        biggest = next;
  }
  va_end(args);
  return biggest;
}
```

# vfprintf

The `vfprintf` function writes formatted output to a stream.

## Definition

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

## Purpose

This function writes a string of characters controlled by the string pointed to by `format` to the file pointed to by `stream`. See the description of the function `printf` for details on the `format` string. The function is equivalent to `fprintf`, except that a pointer to a variable argument list is passed in place of the variable arguments. This pointer should have been initialized using `va_start` (and possibly `va_arg`). A function should not use the variable argument pointer `arg` after calling `vfprintf`, and should call `va_end` before returning.

## Returns

The `vfprintf` function returns the number of characters written to the stream. If a write error occurs, a negative value is returned, and `errno` will be set.

## Related functions

`fprintf, printf, sprintf, vprintf, vsprintf`

**Example**

```
#include <stdio.h>

/* This prints a list of numbers to the stream. If more
   than 5 are to be printed, the first 5 are printed
   followed by etc. */

void ftrunc_seq ( FILE * stream, int n, ...)

{  char format[21];
   va_list arg_ptr;

   strcpy (format, "%d,%d,%d,%d,%d etc.");
   if (n < 5) format[ n * 3] = '\0';
   va_start (arg_ptr, n);
   vfprintf ( stream, format, arg_ptr);
   va_end (arg_ptr);
}
```

# vprintf

The `vprintf` function writes formatted output to standard output.

## Definition

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

## Purpose

This function writes a string of characters controlled by the string pointed to by `format` to standard output. See the description of the function `printf` for details on the `format` string. The function is equivalent to `printf`, except that a pointer to a variable argument list is passed in place of the variable arguments. This pointer should have been initialized using `va_start` (and possibly `va_arg`). A function should not use the variable argument pointer `arg` after calling `vfprintf`, and should call `va_end` before returning.

## Returns

The `vfprintf` function returns the number of characters written to the stream. If a write error occurs, a negative value is returned, and `errno` will be set.

## Related functions

`fprintf, printf, sprintf, vprintf, vsprintf, cprintf`

**Example**

```
#include <stdio.h>


/* This prints a list of numbers to the screen. If more
   than 5 are to be printed, the first 5 are printed
   followed by etc. */

void ftrunc_seq ( int n, ...)

{  char format[21];
   va_list arg_ptr;

   strcpy (format, "%d,%d,%d,%d,%d etc.");
   strcat (format,"\n");
   if (n < 5) format[ n * 3 - 1] = '\0';
   va_start (arg_ptr, n);
   vprintf ( format, arg_ptr);
   va_end (arg_ptr);
 }
```

# vsprintf

The `vsprintf` function writes formatted output to a string.

## Definition

```
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

## Purpose

This function writes a string of characters controlled by the string pointed to by `format` to the string pointed to by `s`. See the description of the function `printf` for details on the `format` string. The function is equivalent to `sprintf`, except that a pointer to a variable argument list is passed in place of the variable arguments. This pointer should have been initialized using `va_start` (and possibly `va_arg`). A function should not use the variable argument pointer `arg` after calling `vsfprintf`, and should call `va_end` before returning.

## Returns

The `vsfprintf` function returns the number of characters written to the string not including the terminating null character.

## Related functions

`fprintf, printf, sprintf, vprintf, vfprintf`

**Example**

```
#include <stdio.h>


/* This prints a list of numbers to a string If more
   than 5 are to be printed, the first 5 are printed
   followed by etc. */

void strunc_seq (char *string, int n, ...)

{  char format[21];
   va_list arg_ptr;

   strcpy (format, "%d,%d,%d,%d,%d etc.");
   if (n < 5) format[ n * 3 - 1] = '\0';
   va_start (arg_ptr, n);
   vsprintf (format, arg_ptr);
   va_end (arg_ptr);
}
```

# write

The `write` function writes data to an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long int write (int handle, void *buffer,
                long int length);
```

## Purpose

This function attempts to write `length` bytes to the file whose handle is given by `handle` from the object pointed to by `buffer`. If the file was opened in text mode, carriage returns will be inserted before each new-line character written – these will not be counted towards the number of bytes written.

## Returns

The `write` function returns the number of bytes written from `buffer` – this may be less than `length` if the disk had insufficient room. If an error occurred, it returns `-1L`, and `errno` will be set to indicate the error.

## Related functions

`open, fread, fwrite, read, _write`

## Example

```
#include <io.h>

char screencopy[32768];
int handle;
unsigned int written;

handle = open ("data", O_WRONLY | O_CREAT, 0);
written = write (handle, screencopy, 32768);

if (written != 32768)
 perror ("Error in saving screencopy");
```

# _write

The `_write` function writes data to an unbuffered file. It is not part of the draft ANSI standard.

## Definition

```
#include <io.h>
long int _write (int handle, void *buffer,
                 long int length);
```

## Purpose

This function attempts to write up to `length` bytes to the file whose handle is given by `handle` from the object pointed to by `buffer`. A direct call is made to GEMDOS, without any translation of carriage returns for text files, and without setting `errno` if errors are detected.

This function is defined as a macro in `io.h`, but if `io.h` is not `#included`, or if `_write` is `#undef`'d, a library function will be called.

## Returns

The `_write` function returns the number of bytes read from `buffer` – this may be less than `length` if the disk had insufficient room. If an error occurs, it returns a negative GEMDOS error code, equal in magnitude to one of the positive error codes defined in `errno.h`.

## Related functions

`open, fread, fwrite, read, _read, write`

**Example**

```
#include <io.h>
int handle1, handle2;
char *fileimage;
long bytes;

/* Duplicate a file */
handle1 = open("file1", O_RDONLY, 0);
handle2 = open("file2", O_WRONLY | O_CREAT | O_TRUNC,
               S_IREAD | S_IWRITE);

bytes = filelength(handle1);
fileimage = malloc (bytes);
if (fileimage != NULL)
{ _read(handle1, fileimage, bytes);
  _write(handle2, fileimage, bytes);
}
else puts("Insufficient memory");

free(fileimage);
```

# xbios

The `xbios` function calls an Atari XBIOS function.

## Definition

```
#include <dos.h>
long int xbios (int funcno, ...);
```

## Purpose

This function is used to make a call to one of the Atari's XBIOS (Extended BIOS) functions. The `funcno` parameter specifies which XBIOS function is required. Other parameters are passed where appropriate – their number, type and purpose depend on the XBIOS function requested. See Atari technical information for further details.

## Returns

The `xbios` function returns the value returned in `d0` by the corresponding XBIOS function.

## Related functions

`bios, gemdos`

## Example

This (unhelpful!) example sets the keyboard delay and repeat rates to an impossible speed.

```
#include <dos.h>

xbios (35, 1, 1);
```

# 4 INDEX OF FUNCTIONS

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS