Watcom C Library Reference

Volume 2

Edition 11.0c

Notice of Copyright

Preface

This manual describes the Watcom C Library. It includes the Standard C Library (as defined in the ANSI C Standard) plus many additional library routines which make application development for personal computers much easier.

Acknowledgements

This book was produced with the Watcom GML electronic publishing system, a software tool developed by WATCOM. In this system, writers use an ASCII text editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The Watcom GML software, which runs on a variety of operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

September, 2000.

Trademarks Used in this Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Synopsis:

```
#include <io.h>
int lock( int handle,
          unsigned long offset,
          unsigned long nbytes );
```

Description: The `lock` function locks *nbytes* amount of data in the file designated by *handle* starting at byte *offset* in the file. This prevents other processes from reading or writing into the locked region until an `unlock` has been done for this locked region of the file.

Multiple regions of a file can be locked, but no overlapping regions are allowed. You cannot unlock multiple regions in the same call, even if the regions are contiguous. All locked regions of a file should be unlocked before closing a file or exiting the program.

With DOS, locking is supported by version 3.0 or later. Note that `SHARE.COM` or `SHARE.EXE` must be installed.

Returns: The `lock` function returns zero if successful, and -1 when an error occurs. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `locking`, `open`, `sopen`, `unlock`

Example:

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

void main()
  {
    int handle;
    char buffer[20];

    handle = open( "file", O_RDWR | O_TEXT );
    if( handle != -1 ) {
      if( lock( handle, 0L, 20L ) ) {
        printf( "Lock failed\n" );
      } else {
        read( handle, buffer, 20 );
        /* update the buffer here */
        lseek( handle, 0L, SEEK_SET );
        write( handle, buffer, 20 );
        unlock( handle, 0L, 20L );
      }
      close( handle );
    }
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <sys\locking.h>
int locking( int handle, int mode, long nbyte );
int _locking( int handle, int mode, long nbyte );
```

Description: The `locking` function locks or unlocks *nbyte* bytes of the file specified by *handle.* Locking a region of a file prevents other processes from reading or writing the locked region until the region has been unlocked. The locking and unlocking takes place at the current file position. The argument *mode* specifies the action to be performed. The possible values for mode are:

Mode	***Meaning***
_LK_LOCK, LK_LOCK	Locks the specified region. The function will retry to lock the region after 1 second intervals until successful or until 10 attempts have been made.
_LK_RLCK, LK_RLCK	Same action as `_LK_LOCK.`
_LK_NBLCK, LK_NBLCK	Non-blocking lock: makes only 1 attempt to lock the specified region.
_LK_NBRLCK, LK_NBRLCK	Same action as `_LK_NBLCK.`
_LK_UNLCK, LK_UNLCK	Unlocks the specified region. The region must have been previously locked.

Multiple regions of a file can be locked, but no overlapping regions are allowed. You cannot unlock multiple regions in the same call, even if the regions are contiguous. All locked regions of a file should be unlocked before closing a file or exiting the program.

With DOS, locking is supported by version 3.0 or later. Note that `SHARE.COM` or `SHARE.EXE` must be installed.

The `_locking` function is identical to `locking`. Use `_locking` for ANSI/ISO naming conventions.

Returns: The `locking` function returns zero if successful. Otherwise, it returns -1 and `errno` is set to indicate the error.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	*Meaning*
EACCES	Indicates a locking violation (file already locked or unlocked).
EBADF	Indicates an invalid file handle.
EDEADLOCK	Indicates a locking violation. This error is returned when *mode* is `LK_LOCK` or `LK_RLCK` and the file cannot be locked after 10 attempts.
EINVAL	Indicates that an invalid argument was given to the function.

See Also: `creat`, `_dos_creat`, `_dos_open`, `lock`, `open`, `sopen`, `unlock`

Example:

```
#include <stdio.h>
#include <sys\locking.h>
#include <share.h>
#include <fcntl.h>
#include <io.h>

void main()
  {
    int handle;
    unsigned nbytes;
    unsigned long offset;
    auto char buffer[512];

    nbytes = 512;
    offset = 1024;
    handle = sopen( "db.fil", O_RDWR, SH_DENYNO );
    if( handle != -1 ) {
      lseek( handle, offset, SEEK_SET );
      locking( handle, LK_LOCK, nbytes );
      read( handle, buffer, nbytes );
      /* update data in the buffer */
      lseek( handle, offset, SEEK_SET );
      write( handle, buffer, nbytes );
      lseek( handle, offset, SEEK_SET );
      locking( handle, LK_UNLCK, nbytes );
      close( handle );
    }
  }
```

Classification: WATCOM

_locking conforms to ANSI/ISO naming conventions

Systems: `locking - All`
`_locking - All`

Synopsis:

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

Description: The `log` function computes the natural logarithm (base e) of *x*. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Returns: The `log` function returns the natural logarithm of the argument. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `EDOM`, and print a "DOMAIN error" diagnostic message using the `stderr` stream.

See Also: `exp`, `log10`, `log2`, `pow`, `matherr`

Example:

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

void main()
  {
    printf( "%f\n", log(.5) );
  }
```

produces the following:

```
-0.693147
```

Classification: ANSI

Systems: Math

Synopsis:

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

Description: The `log10` function computes the logarithm (base 10) of *x*. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Returns: The `log10` function returns the logarithm (base 10) of the argument. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `EDOM`, and print a "DOMAIN error" diagnostic message using the `stderr` stream.

See Also: `exp`, `log`, `log2`, `pow`, `matherr`

Example:

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

void main()
  {
    printf( "%f\n", log10(.5) );
  }
```

produces the following:

```
-0.301030
```

Classification: ANSI

Systems: Math

Synopsis:
```
#include <math.h>
double log2( double x );
```

Description: The `log2` function computes the logarithm (base 2) of *x*. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Returns: The `log2` function returns the logarithm (base 2) of the argument. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `EDOM`, and print a "DOMAIN error" diagnostic message using the `stderr` stream.

See Also: `exp`, `log`, `log10`, `pow`, `matherr`

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

void main()
  {
    printf( "%f\n", log2(.25) );
  }
```

produces the following:

```
-2.000000
```

Classification: WATCOM

Systems: Math

Synopsis:

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

Description: The `longjmp` function restores the environment saved by the most recent call to the `setjmp` function with the corresponding `jmp_buf` argument.

It is generally a bad idea to use `longjmp` to jump out of an interrupt function or a signal handler (unless the signal was generated by the `raise` function).

Returns: After the `longjmp` function restores the environment, program execution continues as if the corresponding call to `setjmp` had just returned the value specified by *return_value*. If the value of *return_value* is 0, the value returned is 1.

See Also: `setjmp`

Example:

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

jmp_buf env;

rtn()
  {
    printf( "about to longjmp\n" );
    longjmp( env, 14 );
  }

void main()
  {
    int ret_val = 293;

    if( 0 == ( ret_val = setjmp( env ) ) ) {
      printf( "after setjmp %d\n", ret_val );
      rtn();
      printf( "back from rtn %d\n", ret_val );
    } else {
      printf( "back from longjmp %d\n", ret_val );
    }
  }
```

produces the following:

```
after setjmp 0
about to longjmp
back from longjmp 14
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <stdlib.h>
unsigned long _lrotl( unsigned long value,
                      unsigned int shift );
```

Description: The `_lrotl` function rotates the unsigned long integer, determined by *value,* to the left by the number of bits specified in *shift.*

Returns: The rotated value is returned.

See Also: `_lrotr`, `_rotl`, `_rotr`

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

unsigned long mask = 0x12345678;

void main()
  {
    mask = _lrotl( mask, 4 );
    printf( "%08lX\n", mask );
  }
```

produces the following:

```
23456781
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
unsigned long _lrotr( unsigned long value,
                      unsigned int shift );
```

Description: The `_lrotr` function rotates the unsigned long integer, determined by *value*, to the right by the number of bits specified in *shift*.

Returns: The rotated value is returned.

See Also: `_lrotl`, `_rotl`, `_rotr`

Example:

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

unsigned long mask = 0x12345678;

void main()
  {
    mask = _lrotr( mask, 4 );
    printf( "%08lX\n", mask );
  }
```

produces the following:

```
81234567
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <search.h>
void *lsearch( const void *key, /* object to search for */
               void *base,      /* base of search data  */
               unsigned *num,   /* number of elements   */
               unsigned width,  /* width of each element*/
               int (*compare)( const void *element1,
                               const void *element2 ) );
```

Description: The `lsearch` function performs a linear search for the value *key* in the array of *num* elements pointed to by *base*. Each element of the array is *width* bytes in size. The argument *compare* is a pointer to a user-supplied routine that will be called by `lsearch` to determine the relationship of an array element with the *key*. One of the arguments to the *compare* function will be an array element, and the other will be *key*.

The *compare* function should return 0 if *element1* is identical to *element2* and non-zero if the elements are not identical.

Returns: If the *key* value is not found in the array, then it is added to the end of the array and the number of elements is incremented. The `lsearch` function returns a pointer to the array element in *base* that matches *key* if it is found, or the newly added key if it was not found.

See Also: `bsearch`, `lfind`

Example:

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

void main( int argc, const char *argv[] )
  {
    int i;
    unsigned num = 0;
    char **array = (char **)calloc( argc, sizeof(char **) );
    extern int compare( const void *, const void * );

    for( i = 1; i < argc; ++i ) {
      lsearch( &argv[i], array, &num, sizeof(char **),
                  compare );
    }
    for( i = 0; i < num; ++i ) {
      printf( "%s\n", array[i] );
    }
  }
```

```
int compare( const void *op1, const void *op2 )
  {
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;
    return( strcmp( *p1, *p2 ) );
  }

/* With input: one two one three four */
```

produces the following:

```
one
two
three
four
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdio.h>
#include <io.h>
long int lseek( int handle, long int offset, int origin );
long int _lseek( int handle, long int offset, int origin );
__int64 _lseeki64( int handle, __int64 offset, int origin );
```

Description: The `lseek` function sets the current file position at the operating system level. The file is referenced using the file handle *handle* returned by a successful execution of one of the `creat`, `dup`, `dup2`, `open` or `sopen` functions. The value of *offset* is used as a relative offset from a file position determined by the value of the argument *origin*.

The new file position is determined in a manner dependent upon the value of *origin* which may have one of three possible values (defined in the `<stdio.h>` header file):

Origin	***Definition***
SEEK_SET	The new file position is computed relative to the start of the file. The value of *offset* must not be negative.
SEEK_CUR	The new file position is computed relative to the current file position. The value of *offset* may be positive, negative or zero.
SEEK_END	The new file position is computed relative to the end of the file.

An error will occur if the requested file position is before the start of the file.

The requested file position may be beyond the end of the file. On POSIX-conforming systems, if data is later written at this point, subsequent reads of data in the gap will return bytes whose value is equal to zero until data is actually written in the gap. On systems such DOS and OS/2 that are not POSIX-conforming, data that are read in the gap have arbitrary values.

Some versions of MS-DOS allow seeking to a negative offset, but it is not recommended since it is not supported by other platforms and may not be supported in future versions of MS-DOS.

The `lseek` function does not, in itself, extend the size of a file (see the description of the `chsize` function).

The `_lseek` function is identical to `lseek`. Use `_lseek` for ANSI/ISO naming conventions.

The _lseeki64 function is identical to `lseek` except that it accepts a 64-bit value for the *offset* argument.

The `lseek` function can be used to obtain the current file position (the `tell` function is implemented in terms of `lseek`). This value can then be used with the `lseek` function to reset the file position to that point in the file:

```
long int file_posn;
int handle;

/* get current file position */
file_posn = lseek( handle, 0L, SEEK_CUR );
  /* or */
file_posn = tell( handle );

/* return to previous file position */
file_posn = lseek( handle, file_posn, SEEK_SET );
```

If all records in the file are the same size, the position of the n'th record can be calculated and read, as illustrated in the example included below. The function in this example assumes records are numbered starting with zero and that *rec_size* contains the size of a record in the file (including the carriage-return character in text files).

Returns: If successful, the current file position is returned in a system-dependent manner. A value of 0 indicates the start of the file.

If an error occurs in `lseek`, (-1L) is returned.

If an error occurs in _ `lseek`i64, (-1I64) is returned.

When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EBADF	The *handle* argument is not a valid file handle.
EINVAL	The *origin* argument is not a proper value, or the resulting file offset would be invalid.

See Also: chsize, close, creat, dup, dup2, eof, exec Functions, fdopen, filelength, fileno, fstat, _grow_handles, isatty, open, read, setmode, sopen, stat, tell, write, umask

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

int read_record( int  handle,
                 long rec_numb,
                 int  rec_size,
                 char *buffer )
  {
    if( lseek( handle, rec_numb * rec_size, SEEK_SET )
         == -1L ) {
      return( -1 );
    }
    return( read( handle, buffer, rec_size ) );
  }

void main()
  {
    int  handle;
    int  size_read;
    char buffer[80];

    /* open a file for input */
    handle = open( "file", O_RDONLY | O_TEXT );
    if( handle != -1 ) {

      /* read a piece of the text */
      size_read = read_record( handle, 1, 80, buffer );

      /* test for error */
      if( size_read == -1 ) {
        printf( "Error reading file\n" );
      } else {
        printf( "%.80s\n", buffer );
      }

      /* close the file */
      close( handle );
    }
  }
```

Classification: lseek is POSIX 1003.1, _lseek is not POSIX _lseek conforms to ANSI/ISO naming conventions

Systems:
```
lseek - All, Netware
_lseek - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_lseeki64 - All
```

Synopsis:

```
#include <stdlib.h>
char *ltoa( long int value,
            char *buffer,
            int radix );
char *_ltoa( long int value,
             char *buffer,
             int radix );
wchar_t *_ltow( long int value,
                wchar_t *buffer,
                int radix );
```

Description: The `ltoa` function converts the binary integer *value* into the equivalent string in base *radix* notation storing the result in the character array pointed to by *buffer*. A null character is appended to the result. The size of *buffer* must be at least 33 bytes when converting values in base 2. The value of *radix* must satisfy the condition:

```
2 <= radix <= 36
```

If *radix* is 10 and *value* is negative, then a minus sign is prepended to the result.

The `_ltoa` function is identical to `ltoa`. Use `_ltoa` for ANSI/ISO naming conventions.

The `_ltow` function is identical to `ltoa` except that it produces a wide-character string (which is twice as long).

Returns: The `ltoa` function returns a pointer to the result.

See Also: `atoi`, `atol`, `itoa`, `sscanf`, `strtol`, `strtoul`, `ultoa`, `utoa`

Example:

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

void print_value( long value )
  {
    int base;
    char buffer[33];

    for( base = 2; base <= 16; base = base + 2 )
      printf( "%2d %s\n", base,
              ltoa( value, buffer, base ) );
  }

void main()
  {
    print_value( 12765L );
  }
```

produces the following:

```
 2 11000111011101
 4 3013131
 6 135033
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification: WATCOM

_ltoa conforms to ANSI/ISO naming conventions

Systems:
```
ltoa - All, Netware
_ltoa - All, Netware
_ltow - All
```

Watcom C Library Reference Volume 2

Synopsis:

```
int main( void );
int main( int argc, const char *argv[] );
int wmain( void );
int wmain( int argc, wchar_t *argv[] );
int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine,
                    int nCmdShow );
int PASCAL wWinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     wcharT *lpszCmdLine,
                     int nCmdShow );
```

Description: `main` is a user-supplied function where program execution begins. The command line to the program is broken into a sequence of tokens separated by blanks and are passed to `main` as an array of pointers to character strings in the parameter *argv*. The number of arguments found is passed in the parameter *argc.* The first element of *argv* will be a pointer to a character string containing the program name. The last element of the array pointed to by *argv* will be a NULL pointer (i.e. *argv[argc]* will be `NULL`). Arguments that contain blanks can be passed to `main` by enclosing them within double quote characters (which are removed from that element in the *argv* vector.

The command line arguments can also be obtained in its original format by using the `getcmd` function.

Alternatively, the `main` function can be declared to return `void` (i.e., no return value). In this case, you will not be able to return an exit code from `main` using a `return` statement; to do so, you must use the `exit` function.

The `wmain` function is a user-defined wide-character version of `main` that operates with wide-character strings. If this function is present in the application, then it will be called by the run-time system startup code (and the `main` function, if present, will not be called).

As with `main`, the `wmain` function can be declared to return `void` and the same considerations will apply.

The `WinMain` function is called by the system as the initial entry point for a Windows-based application. The `wWinMain` function is a wide-character version of `WinMain.`

Parameters	***Meaning***
hInstance	Identifies the current instance of the application.
hPrevInstance	Identifies the previous instance of the application. For an application written for Win32, this parameter is always NULL.
lpszCmdLine	Points to a null-terminated string specifying the command line for the application.
nCmdShow	Specifies how the window is to be shown. This parameter can be one of the following values:

Value	***Meaning***
SW_HIDE	Hides the window and activates another window.
SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the system's list.
SW_RESTORE	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as `SW_SHOWNORMAL`).
SW_SHOW	Activates a window and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates a window and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates a window and displays it as an icon.
SW_SHOWMINNOACTIVE	Displays a window as an icon. The active window remains active.
SW_SHOWNA	Displays a window in its current state. The active window remains active.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The active window remains active.

SW_SHOWNORMAL Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_RESTORE).

The WinMain function initializes an application, and then performs a message retrieval-and-dispatch loop that is the top-level control structure for the remainder of the application's execution. The loop terminates when a WM_QUIT message is received. At that point, WinMain exits the application, returning the value passed in the WM_QUIT message's wParam parameter. If WM_QUIT was received as a result of calling PostQuitMessage, the value of wParam is the value of the PostQuitMessage function's nExitCode parameter.

Returns: The main and wmain functions return an exit code back to the calling program (usually the operating system).

If the WinMain function terminates before entering the message loop, it should return 0. Otherwise, it should terminate when it receives a WM_QUIT message and return the exit value contained in that message's wParam parameter.

See Also: abort, atexit, _bgetcmd, exec Functions, exit, _exit, getcmd, getenv, onexit, putenv, spawn Functions, system

Example:

```
#include <stdio.h>

int main( int argc, char *argv[] )
  {
    int i;
    for( i = 0; i < argc; ++i ) {
      printf( "argv[%d] = %s\n", i, argv[i] );
    }
    return( 0 );
  }
#ifdef _WIDE_
int wmain( int wargc, wchar_t *wargv[] )
  {
    int i;
    for( i = 0; i < wargc; ++i ) {
      wprintf( L"wargv[%d] = %s\n", i, wargv[i] );
    }
    return( 0 );
  }
#endif
```

produces the following:

```
argv[0] = C:\WATCOM\DEMO\MYPGM.EXE
argv[1] = hhhhh
argv[2] = another arg
```

when the program `mypgm` is executed with the command

```
mypgm hhhhh  "another arg"
```

A sample Windows main program is shown below.

```
int PASCAL WinMain( HANDLE this_inst, HANDLE prev_inst,
                    LPSTR cmdline, int cmdshow )
  {
    MSG         msg;

    if( !prev_inst ) {
      if( !FirstInstance( this_inst ) ) return( 0 );
    }
    if( !AnyInstance( this_inst, cmdshow ) ) return( 0 );
    /*
      GetMessage returns FALSE when WM_QUIT is received
    */
    while( GetMessage( &msg, NULL, NULL, NULL ) ) {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
    return( msg.wParam );
  }
```

Classification: main is ANSI, wmain is not ANSI, WinMain is not ANSI, wWinMain is not ANSI

Systems:
```
main - All, Netware
wmain - Win32, OS/2-32
WinMain - Windows, Win386, Win32
wWinMain - Win32
```

Synopsis:

```
#include <stdlib.h>
void _makepath( char *path,
                const char *drive,
                const char *dir,
                const char *fname,
                const char *ext );
void _wmakepath( wchar_t *path,
                 const wchar_t *drive,
                 const wchar_t *dir,
                 const wchar_t *fname,
                 const wchar_t *ext );
```

Description: The `_makepath` function constructs a full pathname from the components consisting of a drive letter, directory path, file name and file name extension. The full pathname is placed in the buffer pointed to by the argument *path.*

The `_wmakepath` function is a wide-character version of `_makepath` that operates with wide-character strings.

The maximum size required for each buffer is specified by the manifest constants `_MAX_PATH`, `_MAX_DRIVE`, `_MAX_DIR`, `_MAX_FNAME`, and `_MAX_EXT` which are defined in `<stdlib.h>.`

drive The *drive* argument points to a buffer containing the drive letter (A, B, C, etc.) followed by an optional colon. The `_makepath` function will automatically insert a colon in the full pathname if it is missing. If *drive* is a NULL pointer or points to an empty string, no drive letter or colon will be placed in the full pathname.

dir The *dir* argument points to a buffer containing just the pathname. Either forward slashes (/) or backslashes (\) may be used. The trailing slash is optional. The `_makepath` function will automatically insert a trailing slash in the full pathname if it is missing. If *dir* is a NULL pointer or points to an empty string, no slash will be placed in the full pathname.

fname The *fname* argument points to a buffer containing the base name of the file without any extension (suffix).

ext The *ext* argument points to a buffer containing the filename extension or suffix. A leading period (.) is optional. The `_makepath` routine will automatically insert a period in the full pathname if it is missing. If *ext* is a NULL pointer or points to an empty string, no period will be placed in the full pathname.

Returns: The `_makepath` function returns no value.

See Also: `_fullpath`, `_splitpath`

Example:

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

void main()
  {
    char full_path[ _MAX_PATH ];
    char drive[ _MAX_DRIVE ];
    char dir[ _MAX_DIR ];
    char fname[ _MAX_FNAME ];
    char ext[ _MAX_EXT ];

    _makepath(full_path,"c","watcomc\\h\\","stdio","h");
    printf( "Full path is: %s\n\n", full_path );
    _splitpath( full_path, drive, dir, fname, ext );
    printf( "Components after _splitpath\n" );
    printf( "drive: %s\n", drive );
    printf( "dir:   %s\n", dir );
    printf( "fname: %s\n", fname );
    printf( "ext:   %s\n", ext );
  }
```

produces the following:

```
Full path is: c:watcomc\h\stdio.h

Components after _splitpath
drive: c:
dir:   watcomc\h\
fname: stdio
ext:   .h
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: WATCOM

Systems:

```
_makepath - All, Netware
_wmakepath - All
```

Synopsis:

```
#include <stdlib.h>  For ANSI compatibility (malloc only)
#include <malloc.h>  Required for other function prototypes
void *malloc( size_t size );
void __based(void) *_bmalloc( __segment seg, size_t size );
void __far  *_fmalloc( size_t size );
void __near *_nmalloc( size_t size );
```

Description: The `malloc` functions allocate space for an object of *size* bytes. Nothing is allocated when the *size* argument has a value of zero.

Each function allocates memory from a particular heap, as listed below:

Function	***Heap***
malloc	Depends on data model of the program
_bmalloc	Based heap specified by *seg* value
_fmalloc	Far heap (outside the default data segment)
_nmalloc	Near heap (inside the default data segment)

In a small data memory model, the `malloc` function is equivalent to the `_nmalloc` function; in a large data memory model, the `malloc` function is equivalent to the `_fmalloc` function.

Returns: The `malloc` functions return a pointer to the start of the allocated memory. The `malloc`, `_fmalloc` and `_nmalloc` functions return `NULL` if there is insufficient memory available or if the requested size is zero. The `_bmalloc` function returns `_NULLOFF` if there is insufficient memory available or if the requested size is zero.

See Also: `calloc` Functions, `_expand` Functions, `free` Functions, `halloc`, `hfree`, `_msize` Functions, `realloc` Functions, `sbrk`

Example:

```
#include <stdlib.h>

void main()
  {
    char *buffer;
```

```
            buffer = (char *)malloc( 80 );
            if( buffer != NULL ) {

                /* body of program */

                free( buffer );
            }
          }
```

Classification: malloc is ANSI, _fmalloc is not ANSI, _bmalloc is not ANSI, _nmalloc is not ANSI

Systems:

```
malloc - All, Netware
_bmalloc - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_fmalloc - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_nmalloc - DOS, Windows, Win386, Win32, QNX, OS/2 1.x, OS/2
1.x(MT), OS/2-32
```

Synopsis:
```
#include <math.h>
int matherr( struct _exception *err_info );
```

Description: The `matherr` function is invoked each time an error is detected by functions in the math library. The default `matherr` function supplied in the library returns zero which causes an error message to be displayed upon `stderr` and `errno` to be set with an appropriate error value. An alternative version of this function can be provided, instead of the library version, in order that the error handling for mathematical errors can be handled by an application.

A program may contain a user-written version of `matherr` to take any appropriate action when an error is detected. When zero is returned, an error message will be printed upon `stderr` and `errno` will be set as was the case with the default function. When a non-zero value is returned, no message is printed and `errno` is not changed. The value `err_info->retval` is used as the return value for the function in which the error was detected.

The `matherr` function is passed a pointer to a structure of type `struct _exception` which contains information about the error that has been detected:

```
struct _exception
{ int type;      /* TYPE OF ERROR                */
  char *name;    /* NAME OF FUNCTION             */
  double arg1;   /* FIRST ARGUMENT TO FUNCTION   */
  double arg2;   /* SECOND ARGUMENT TO FUNCTION  */
  double retval; /* DEFAULT RETURN VALUE         */
};
```

The `type` field will contain one of the following values:

Value	***Meaning***
DOMAIN	A domain error has occurred, such as `sqrt(-1e0).`
SING	A singularity will result, such as `pow(0e0,-2).`
OVERFLOW	An overflow will result, such as `pow(10e0,100).`
UNDERFLOW	An underflow will result, such as `pow(10e0,-100).`
TLOSS	Total loss of significance will result, such as `exp(1000).`
PLOSS	Partial loss of significance will result, such as `sin(10e70).`

The `name` field points to a string containing the name of the function which detected the error. The fields `arg1` and `arg2` (if required) give the values which caused the error. The field `retval` contains the value which will be returned by the function. This value may be changed by a user-supplied version of the `matherr` function.

Returns: The `matherr` function returns zero when an error message is to be printed and a non-zero value otherwise.

Example:

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

/* Demonstrate error routine in which negative */
/* arguments to "sqrt" are treated as positive */

void main()
  {
    printf( "%e\n", sqrt( -5e0 ) );
    exit( 0 );
  }

int matherr( struct _exception *err )
  {
    if( strcmp( err->name, "sqrt" ) == 0 ) {
      if( err->type == DOMAIN ) {
        err->retval = sqrt( -(err->arg1) );
        return( 1 );
      } else
        return( 0 );
    } else
      return( 0 );
  }
```

Classification: WATCOM

Systems: Math

Synopsis:
```
#include <stdlib.h>
#define max(a,b)  (((a) > (b)) ? (a) : (b))
```

Description: The `max` macro will evaluate to be the greater of two values. It is implemented as follows.

```
#define max(a,b)  (((a) > (b)) ? (a) : (b))
```

Returns: The `max` macro will evaluate to the larger of the two values passed.

See Also: `min`

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

void main()
  {
    int a;

    /*
     * The following line will set the variable "a" to 10
     * since 10 is greater than 1.
     */
    a = max( 1, 10 );
    printf( "The value is: %d\n", a );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <mbstring.h>
unsigned int _mbbtombc( unsigned int ch );
```

Description: The `_mbbtombc` function returns the double-byte character equivalent to the single-byte character *ch*. The single-byte character must be in the range 0x20 through 0x7E or 0xA1 through 0xDF.

Note: This function was called `hantozen` in earlier versions.

Returns: The `_mbbtombc` function returns *ch* if there is no equivalent double-byte character; otherwise `_mbbtombc` returns a double-byte character.

See Also: `_getmbcp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbctombb`, `_ismbbalnum`, `_ismbbalpha`, `_ismbbgraph`, `_ismbbkalnum`, `_ismbbkalpha`, `_ismbbkana`, `_ismbbkprint`, `_ismbbkpunct`, `_ismbblead`, `_ismbbprint`, `_ismbbpunct`, `_ismbbtrail`, `_mbcjistojms`, `_mbcjmstojis`, `_mbctombb`, `_mbbtype`, `_setmbcp`

Example:

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

char alphabet[] = {
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
};

void main()
  {
    int             i;
    unsigned short  c;

    _setmbcp( 932 );
    for( i = 0; i < sizeof( alphabet ) - 1; i++ ) {
      c = _mbbtombc( alphabet[ i ] );
      printf( "%c%c", c>>8, c );
    }
    printf( "\n" );
  }
```

produces the following:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
#include <mbctype.h> (for manifest constants)
int _mbbtype( unsigned char ch, int type );
```

Description: The `_mbbtype` function determines the type of a byte in a multibyte character. If the value of *type* is any value except 1, `_mbbtype` tests for a valid single-byte or lead byte of a multibyte character. If the value of *type* is 1, `_mbbtype` tests for a valid trail byte of a multibyte character.

Note: A similar function was called `chkctype` in earlier versions.

Returns: If the value of *type* is not 1, the `_mbbtype` function returns one of the following values:

_MBC_SINGLE	the character is a valid single-byte character (e.g., 0x20 - 0x7E, 0xA1 - 0xDF in code page 932)
_MBC_LEAD	the character is valid lead byte character (e.g., 0x81 - 0x9F, 0xE0 - 0xFC in code page 932)
_MBC_ILLEGAL	the character is an illegal character (e.g., any value except 0x20 - 0x7E, 0xA1 - 0xDF, 0x81 - 0x9F, 0xE0 - 0xFC in code page 932)

If the value of *type* is 1, the `_mbbtype` function returns one of the following values:

_MBC_TRAIL	the character is a valid trailing byte character (e.g., 0x40 - 0x7E, 0x80 - 0xFC in code page 932)
_MBC_ILLEGAL	the character is an illegal character (e.g., any character except a valid trailing byte character)

See Also: `_getmbcp`, `_ismbcalnum`, `_ismbcalpha`, `_ismbccntrl`, `_ismbcdigit`, `_ismbcgraph`, `_ismbchira`, `_ismbckata`, `_ismbcl0`, `_ismbcl1`, `_ismbcl2`, `_ismbclegal`, `_ismbclower`, `_ismbcprint`, `_ismbcpunct`, `_ismbcspace`, `_ismbcsymbol`, `_ismbcupper`, `_ismbcxdigit`, `_setmbcp`

Example:

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

const char *types[4] = {
    "ILLEGAL",
    "SINGLE",
    "LEAD",
    "TRAIL"
};

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int     i, j, k;

    _setmbcp( 932 );
    k = 0;
    for( i = 0; i < SIZE; i++ ) {
      j = _mbbtype( chars[i], k );
      printf( "%s\n", types[ 1 + j ] );
      if( j == _MBC_LEAD )
        k = 1;
      else
        k = 0;
    }
  }
```

produces the following:

```
SINGLE
SINGLE
SINGLE
SINGLE
LEAD
TRAIL
LEAD
TRAIL
LEAD
TRAIL
LEAD
TRAIL
SINGLE
SINGLE
SINGLE
LEAD
TRAIL
ILLEGAL
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
int _mbccmp( const unsigned char *s1,
             const unsigned char *s2 );
int _fmbccmp( const unsigned char __far *s1,
              const unsigned char __far *s2 );
```

Description: The `_mbccmp` function compares one multibyte character from *s1* to one multibyte character from *s2.*

The function is a data model independent form of the `_mbccmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbccmp` and functions return the following values.

Value	***Meaning***
< ***0***	multibyte character at *s1* less than multibyte character at *s2*
0	multibyte character at *s1* identical to multibyte character at *s2*
> ***0***	multibyte character at *s1* greater than multibyte character at *s2*

See Also: `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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

unsigned char mb1[2] = {
    0x81, 0x43
};

unsigned char mb2[2] = {
    0x81, 0x42
};

void main()
  {
    int     i;

    _setmbcp( 932 );
    i = _mbccmp( mb1, mb2 );
    if( i < 0 )
        printf( "Less than\n" );
    else if( i == 0 )
        printf( "Equal to\n" );
    else
        printf( "Greater than\n" );

  }
```

produces the following:

```
Greater than
```

Classification: _mbccmp is ANSI, _mbccmp is not ANSI, _fmbccmp is not ANSI

Systems:
```
_mbccmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbccmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <mbstring.h>
void _mbccpy( unsigned char *dest,
              const unsigned char *ch );
void _fmbccpy( unsigned char __far *dest,
               const unsigned char __far *ch );
```

Description: The _mbccpy function copies one multibyte character from *ch* to *dest.*

The function is a data model independent form of the _mbccpy function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The _mbccpy function does not return a value.

See Also: _mbccmp, _mbcicmp, _mbcjistojms, _mbcjmstojis, _mbclen, _mbctohira, _mbctokata, _mbctolower, _mbctombb, _mbctoupper, mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, sisinit, wcrtomb, wcsrtombs, wcstombs, wctob, wctomb

Example:

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

unsigned char mb1[2] = {
    0x00, 0x00
};

unsigned char mb2[4] = {
    0x81, 0x42, 0x81, 0x41
};

void main()
  {
    _setmbcp( 932 );
    printf( "%#6.4x\n", mb1[0] << 8 | mb1[1] );
    _mbccpy( mb1, mb2 );
    printf( "%#6.4x\n", mb1[0] << 8 | mb1[1] );
  }
```

produces the following:

```
  0000
0x8142
```

Classification: WATCOM

Systems:
```
_mbccpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbccpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <mbstring.h>
int _mbcicmp( const unsigned char *s1,
              const unsigned char *s2 );
int _fmbcicmp( const unsigned char __far *s1,
               const unsigned char __far *s2 );
```

Description: The `_mbcicmp` function compares one multibyte character from *s1* to one multibyte character from *s2* using a case-insensitive comparison.

The function is a data model independent form of the `_mbcicmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbcicmp` and functions return the following values.

Value	***Meaning***
< ***0***	multibyte character at *s1* less than multibyte character at *s2*
0	multibyte character at *s1* identical to multibyte character at *s2*
> ***0***	multibyte character at *s1* greater than multibyte character at *s2*

See Also: `_mbccmp`, `_mbccpy`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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

unsigned char mb1[2] = {
    0x41, 0x42
};

unsigned char mb2[2] = {
    0x61, 0x43
};

void main()
  {
    int     i;

    _setmbcp( 932 );
    i = _mbcicmp( mb1, mb2 );
    if( i < 0 )
        printf( "Less than\n" );
    else if( i == 0 )
        printf( "Equal to\n" );
    else
        printf( "Greater than\n" );

  }
```

produces the following:

```
Equal to
```

Classification: WATCOM

Systems: _mbcicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbcicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <mbstring.h>
unsigned int _mbcjistojms( unsigned int ch );
```

Description: The _mbcjistojms converts a JIS character set code to a shift-JIS character set code. If the argument is out of range, _mbcjistojms returns 0. Valid JIS double-byte characters are those in which the first and second byte fall in the range 0x21 through 0x7E. This is summarized in the following diagram.

```
[ 1st byte ]    [ 2nd byte ]
 0x21-0x7E       0x21-0x7E
```

Note: The JIS character set code is a double-byte character set defined by JIS, the Japan Industrial Standard Institutes. Shift-JIS is another double-byte character set. It is defined by Microsoft for personal computers and is based on the JIS code. The first byte and the second byte of JIS codes can have values less than 0x80. Microsoft has designed shift-JIS code so that it can be mixed in strings with single-byte alphanumeric codes. Thus the double-byte shift-JIS codes are greater than or equal to 0x8140.

Note: This function was called jistojms in earlier versions.

Returns: The _mbcjistojms function returns zero if the argument is not in the range otherwise, the corresponding shift-JIS code is returned.

See Also: _getmbcp, _mbbtombc, _mbcjmstojis, _mbctombb, _ismbbalnum, _ismbbalpha, _ismbbgraph, _ismbbkalnum, _ismbbkalpha, _ismbbkana, _ismbbkprint, _ismbbkpunct, _ismbblead, _ismbbprint, _ismbbpunct, _ismbbtrail, _mbbtombc, _mbcjmstojis, _mbctombb, _mbbtype, _setmbcp

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

void main()
  {
    unsigned short c;

    _setmbcp( 932 );
    c = _mbcjistojms( 0x2152 );
    printf( "%#6.4x\n", c );
  }
```

produces the following:

```
0x8171
```

Classification: WATCOM

Systems: All

Synopsis:

```
#include <mbstring.h>
unsigned int _mbcjmstojis( unsigned int ch );
```

Description: The `_mbcjmstojis` converts a shift-JIS character set code to a JIS character set code. If the argument is out of range, `_mbcjmstojis` returns 0. Valid shift-JIS double-byte characters are those in which the first byte falls in the range 0x81 through 0x9F or 0xE0 through 0xFC and whose second byte falls in the range 0x40 through 0x7E or 0x80 through 0xFC. This is summarized in the following diagram.

```
[ 1st byte ]    [ 2nd byte ]
 0x81-0x9F       0x40-0xFC
    or           except 0x7F
 0xE0-0xFC
```

Note: The JIS character set code is a double-byte character set defined by JIS, the Japan Industrial Standard Institutes. Shift-JIS is another double-byte character set. It is defined by Microsoft for personal computers and is based on the JIS code. The first byte and the second byte of JIS codes can have values less than 0x80. Microsoft has designed shift-JIS code so that it can be mixed in strings with single-byte alphanumeric codes. Thus the double-byte shift-JIS codes are greater than or equal to 0x8140.

Note: This function was called `jmstojis` in earlier versions.

Returns: The `_mbcjmstojis` function returns zero if the argument is not in the range otherwise, the corresponding shift-JIS code is returned.

See Also: `_getmbcp`, `_mbbtombc`, `_mbcjistojms`, `_mbctombb`, `_ismbbalnum`, `_ismbbalpha`, `_ismbbgraph`, `_ismbbkalnum`, `_ismbbkalpha`, `_ismbbkana`, `_ismbbkprint`, `_ismbbkpunct`, `_ismbblead`, `_ismbbprint`, `_ismbbpunct`, `_ismbbtrail`, `_mbbtombc`, `_mbcjistojms`, `_mbctombb`, `_mbbtype`, `_setmbcp`

Example:

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

void main()
  {
    unsigned short c;

    _setmbcp( 932 );
    c = _mbcjmstojis( 0x8171 );
    printf( "%#6.4x\n", c );
  }
```

produces the following:

```
0x2152
```

Classification: WATCOM

Systems: All

Synopsis:
```
#include <mbstring.h>
size_t _mbclen( const unsigned char *ch );
size_t far _fmbclen( const unsigned char __far *ch );
```

Description: The `_mbclen` function determines the number of bytes comprising the multibyte character pointed to by *ch*.

The function is a data model independent form of the `_mbclen` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: If *ch* is a NULL pointer, the `_mbclen` function returns zero if multibyte character encodings do not have state-dependent encoding, and non-zero otherwise. If *ch* is not a NULL pointer, the `_mbclen` function returns:

Value	***Meaning***
0	if *ch* points to the null character
1	if *ch* points to a single-byte character
2	if *ch* points to a double-byte character
-1	if *ch* does not point to a valid multibyte character

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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

unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00       /* null character */
};

void main()
  {
    int     i, j;

    _setmbcp( 932 );
    for( i = 0; i < sizeof(chars); i += j ) {
        j = _mbclen( &chars[i] );
        printf( "%d bytes in character\n", j );
    }
  }
```

produces the following:

```
1 bytes in character
1 bytes in character
1 bytes in character
1 bytes in character
2 bytes in character
2 bytes in character
2 bytes in character
2 bytes in character
1 bytes in character
1 bytes in character
1 bytes in character
2 bytes in character
1 bytes in character
```

Classification: WATCOM

Systems:
```
_mbclen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbclen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <mbstring.h>
unsigned int _mbctolower( unsigned int c );
```

Description: The `_mbctolower` function converts an uppercase multibyte character to an equivalent lowercase multibyte character.

For example, in code page 932, this includes the single-byte uppercase letters A-Z and the double-byte uppercase characters such that:

```
    0x8260 <= c <= 0x8279
```

Note: This function was called `jtolower` in earlier versions.

Returns: The `_mbctolower` function returns the argument value if the argument is not a double-byte uppercase character; otherwise, the equivalent lowercase character is returned.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

unsigned int chars[] = {
    'A',        /* single-byte A */
    'B',        /* single-byte B */
    'C',        /* single-byte C */
    'D',        /* single-byte D */
    'E',        /* single-byte E */
    0x8260,     /* double-byte A */
    0x8261,     /* double-byte B */
    0x8262,     /* double-byte C */
    0x8263,     /* double-byte D */
    0x8264      /* double-byte E */
};

#define SIZE sizeof( chars ) / sizeof( unsigned int )

void main()
  {
    int   i;
    unsigned int c;
    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      c = _mbctolower( chars[ i ] );
      if( c > 0xff )
        printf( "%c%c", c>>8, c );
      else
        printf( "%c", c );
    }
    printf( "\n" );
  }
```

produces the following:

```
abcde a b c d e
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <mbstring.h>
unsigned int _mbctoupper( unsigned int c );
```

Description: The `_mbctoupper` function converts a lowercase multibyte character to an equivalent uppercase multibyte character.

For example, in code page 932, this includes the single-byte lowercase letters a-z and the double-byte lowercase characters such that:

```
    0x8281 <= c <= 0x829A
```

Note: This function was called `jtoupper` in earlier versions.

Returns: The `_mbctoupper` function returns the argument value if the argument is not a double-byte lowercase character; otherwise, the equivalent uppercase character is returned.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

unsigned int chars[] = {
    'a',        /* single-byte a */
    'b',        /* single-byte b */
    'c',        /* single-byte c */
    'd',        /* single-byte d */
    'e',        /* single-byte e */
    0x8281,     /* double-byte a */
    0x8282,     /* double-byte b */
    0x8283,     /* double-byte c */
    0x8284,     /* double-byte d */
    0x8285      /* double-byte e */
};

#define SIZE sizeof( chars ) / sizeof( unsigned int )

void main()
  {
    int   i;
    unsigned int c;
    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      c = _mbctoupper( chars[ i ] );
      if( c > 0xff )
        printf( "%c%c", c>>8, c );
      else
        printf( "%c", c );
    }
    printf( "\n" );
  }
```

produces the following:

```
ABCDE A B C D E
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
unsigned int _mbctohira( unsigned int ch );
```

Description: The _mbctohira converts a double-byte Katakana character to a Hiragana character. A double-byte Katakana character is any character for which the following expression is true:

```
    0x8340 <= ch <= 0x8396  &&  ch != 0x837F
```

Any Katakana character whose value is less than 0x8393 is converted to Hiragana (there are 3 extra Katakana characters that have no equivalent).

Note: The Japanese double-byte character set includes Kanji, Hiragana, and Katakana characters - both alphabetic and numeric. Kanji is the ideogram character set of the Japanese character set. Hiragana and Katakana are two types of phonetic character sets of the Japanese character set. The Hiragana code set includes 83 characters and the Katakana code set includes 86 characters.

Note: This function was called jtohira in earlier versions.

Returns: The _mbctohira function returns the argument value if the argument is not a double-byte Katakana character; otherwise, the equivalent Hiragana character is returned.

See Also: _mbccmp, _mbccpy, _mbcicmp, _mbcjistojms, _mbcjmstojis, _mbclen, _mbctokata, _mbctolower, _mbctombb, _mbctoupper, mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, sisinit, wcrtomb, wcsrtombs, wcstombs, wctob, wctomb

Example:

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

unsigned int chars[] = {
    0x8340,
    0x8364,
    0x8396
};
```

```
#define SIZE sizeof( chars ) / sizeof( unsigned int )

void main()
  {
    int   i;

    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      printf( "%#6.4x - %#6.4x\n",
              chars[ i ],
              _mbctohira( chars[ i ] ) );
    }
  }
```

produces the following:

```
0x8340 - 0x829f
0x8364 - 0x82c3
0x8396 - 0x8396
```

Classification: WATCOM

Systems: All

Synopsis:
```
#include <mbstring.h>
unsigned int _mbctokata( unsigned int ch );
```

Description: The `_mbctokata` converts a double-byte Hiragana character to a Katakana character. A double-byte Hiragana character is any character for which the following expression is true:

```
    0x829F <= c <= 0x82F1
```

Note: The Japanese double-byte character set includes Kanji, Hiragana, and Katakana characters - both alphabetic and numeric. Kanji is the ideogram character set of the Japanese character set. Hiragana and Katakana are two types of phonetic character sets of the Japanese character set. The Hiragana code set includes 83 characters and the Katakana code set includes 86 characters.

Note: This function was called `jtokata` in earlier versions.

Returns: The `_mbctokata` function returns the argument value if the argument is not a double-byte Hiragana character; otherwise, the equivalent Katakana character is returned.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

unsigned int chars[] = {
    0x829F,
    0x82B0,
    0x82F1
};
```

```
#define SIZE sizeof( chars ) / sizeof( unsigned int )

void main()
  {
    int   i;

    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      printf( "%#6.4x - %#6.4x\n",
              chars[ i ],
              _mbctokata( chars[ i ] ) );
    }
  }
```

produces the following:

```
0x829f - 0x8340
0x82b0 - 0x8351
0x82f1 - 0x8393
```

Classification: WATCOM

Systems: All

Synopsis:

```
#include <mbstring.h>
unsigned int _mbctombb( unsigned int ch );
```

Description: The `_mbctombb` function returns the single-byte character equivalent to the double-byte character *ch*. The single-byte character will be in the range 0x20 through 0x7E or 0xA1 through 0xDF.

Note: This function was called `zentohan` in earlier versions.

Returns: The `_mbctombb` function returns *ch* if there is no equivalent single-byte character; otherwise `_mbctombb` returns a single-byte character.

See Also: `_getmbcp`, `_mbbtombc`, `_mbcjistojms`, `_mbcjmstojis`, `_ismbbalnum`, `_ismbbalpha`, `_ismbbgraph`, `_ismbbkalnum`, `_ismbbkalpha`, `_ismbbkana`, `_ismbbkprint`, `_ismbbkpunct`, `_ismbblead`, `_ismbbprint`, `_ismbbpunct`, `_ismbbtrail`, `_mbbtombc`, `_mbcjistojms`, `_mbcjmstojis`, `_mbbtype`, `_setmbcp`

Example:

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

#define ZEN(x) 130*256+(x-1+32)

unsigned int alphabet[26] = {
    ZEN('A'),ZEN('B'),ZEN('C'),ZEN('D'),ZEN('E'),
    ZEN('F'),ZEN('G'),ZEN('H'),ZEN('I'),ZEN('J'),
    ZEN('K'),ZEN('L'),ZEN('M'),ZEN('N'),ZEN('O'),
    ZEN('P'),ZEN('Q'),ZEN('R'),ZEN('S'),ZEN('T'),
    ZEN('U'),ZEN('V'),ZEN('W'),ZEN('X'),ZEN('Y'),
    ZEN('Z')
};

#define SIZE sizeof( alphabet ) / sizeof( unsigned int )

void main()
  {
    int             i;
    unsigned int    c;

    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      c = _mbctombb( alphabet[ i ] );
      printf( "%c", c );
    }
    printf( "\n" );
  }
```

produces the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbgetcode( unsigned char *mbstr,
                           unsigned int *dbchp );
unsigned char far *_fmbgetcode( unsigned char far *mbstr,
                                unsigned int *dbchp );
```

Description: The `_mbgetcode` function places the next single- or double-byte character from the start of the Kanji string specified by *mbstr* in the wide character pointed to by *dbchp*. If the second-half of a double-byte character is NULL, then the returned wide character is NULL.

The function is a code and data model independent form of the `_mbgetcode` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `_mbgetcode` function returns a pointer to the next character to be obtained from the string. If *mbstr* points at a null character then *mbstr* is returned.

See Also: `_mbsnccnt`, `_mbputchar`

Example:

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

unsigned char set[] = {
    "ab\x81\x41\x81\x42\cd\x81"
};

void main()
  {
    unsigned int c;
    unsigned char *str;

    _setmbcp( 932 );
    str = set;
    for( ; *str != '\0'; ) {
        str = _mbgetcode( str, &c );
        printf( "Character code 0x%2.2x\n", c );
    }
  }
```

produces the following:

```
Character code 0x61
Character code 0x62
Character code 0x8141
Character code 0x8142
Character code 0x63
Character code 0x64
Character code 0x00
```

Classification: WATCOM

Systems: _mbgetcode - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbgetcode - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <stdlib.h>
    or
#include <mbstring.h>
int mblen( const char *s, size_t n );
int _fmblen( const char __far *s, size_t n );
```

Description: The `mblen` function determines the number of bytes comprising the multibyte character pointed to by *s*. At most *n* bytes of the array pointed to by *s* will be examined.

The `_fmblen` function is a data model independent form of the `mblen` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: If *s* is a NULL pointer, the `mblen` function returns zero if multibyte character encodings are not state dependent, and non-zero otherwise. If *s* is not a NULL pointer, the `mblen` function returns:

Value	***Meaning***
0	if *s* points to the null character
len	the number of bytes that comprise the multibyte character (if the next *n* or fewer bytes form a valid multibyte character)
-1	if the next *n* bytes do not form a valid multibyte character

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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


const char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    int         i, j, k;

    _setmbcp( 932 );
    printf( "Character encodings are %sstate dependent\n",
            ( mblen( NULL, MB_CUR_MAX ) ) ? "" : "not " );
    j = 1;
    for( i = 0; j > 0; i += j ) {
      j = mblen( &chars[i], MB_CUR_MAX );
      printf( "%d bytes in character ", j );
      if( j == 0 ) {
        k = 0;
      } else if ( j == 1 ) {
        k = chars[i];
      } else if( j == 2 ) {
        k = chars[i]<<8 | chars[i+1];
      }
      printf( "(%#6.4x)\n", k );
    }
  }
```

produces the following:

```
Character encodings are not state dependent
1 bytes in character (0x0020)
1 bytes in character (0x002e)
1 bytes in character (0x0031)
1 bytes in character (0x0041)
2 bytes in character (0x8140)
2 bytes in character (0x8260)
2 bytes in character (0x82a6)
2 bytes in character (0x8342)
1 bytes in character (0x00a1)
1 bytes in character (0x00a6)
1 bytes in character (0x00df)
2 bytes in character (0xe0a1)
0 bytes in character (  0000)
```

Classification: mblen is ANSI, _fmblen is not ANSI

Systems:
```
mblen - All, Netware
_fmblen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbputchar( unsigned char *mbstr,
                           unsigned int dbch );
unsigned char far *_fmbputchar( unsigned char far *mbstr,
                                unsigned int dbch );
```

Description: The `_mbputchar` function places the next single- or double-byte character specified by *dbch* at the start of the buffer specified by *mbstr*.

The function is a code and data model independent form of the `_mbputchar` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `_mbputchar` function returns a pointer to the next location in which to store a character.

See Also: `_mbsnccnt`, `_mbgetcode`

Example:

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

void main()
  {
    unsigned int c;
    unsigned char *str1;
    unsigned char *str2;
    unsigned char buf[30];

    _setmbcp( 932 );
    str1 = "ab\x82\x62\x82\x63\ef\x81\x66";
    str2 = buf;

    for( ; *str1 != '\0'; ) {
        str1 = _mbgetcode( str1, &c );
        str2 = _mbputchar( str2, '<' );
        str2 = _mbputchar( str2, c );
        str2 = _mbputchar( str2, '>' );
    }
    *str2 = '\0';
    printf( "%s\n", buf );
  }
```

produces the following:

```
<a><b>< C>< D><e><f>< G>
```

Classification: WATCOM

Systems:

```
_mbputchar - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbputchar - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <wchar.h>
int mbrlen( const char *s, size_t n, mbstate_t *ps );
int _fmbrlen( const char far *s, size_t n, mbstate_t far *ps )
;
```

Description: The `mbrlen` function determines the number of bytes comprising the multibyte character pointed to by *s*. The `mbrlen` function is equivalent to the following call:

```
mbrtowc((wchar_t *)0, s, n, ps != 0 ? ps : &internal)
```

where `&internal` is the address of the internal `mbstate_t` object for the `mbrlen` function.

The `_fmbrlen` function is a data model independent form of the `mbrlen` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The restartable multibyte/wide character conversion functions differ from the corresponding internal-state multibyte character functions (`mblen`, `mbtowc`, and `wctomb`) in that they have an extra argument, *ps*, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead. You are guaranteed that no other function in the library calls these functions with a null pointer for *ps*, thereby ensuring the stability of the state.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

If the encoding is state-dependent, on entry each function takes the described conversion state (either internal or pointed to by *ps*) as current. The conversion state described by the pointed-to object is altered as needed to track the shift state of the associated multibyte character sequence. For encodings without state dependency, the pointer to the `mbstate_t` argument is ignored.

Returns: The `mbrlen` function returns a value between -2 and *n*, inclusive. The `mbrlen` function returns the first of the following that applies:

Value	***Meaning***
0	if the next *n* or fewer bytes form the multibyte character that corresponds to the null wide character.
>0	if the next *n* or fewer bytes form a valid multibyte character; the value returned is the number of bytes that constitute that multibyte character.

-2 if the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed; it is unspecified whether this can occur when the value of *n* is less than that of the `MB_CUR_MAX` macro.

-1 if an encoding error occurs (when the next *n* or fewer bytes do not form a complete and valid multibyte character); the value of the macro `EILSEQ` will be stored in `errno,` but the conversion state will be unchanged.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>


const char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};
```

```
void main()
  {
    int         i, j, k;

    _setmbcp( 932 );
    j = 1;
    for( i = 0; j > 0; i += j ) {
      j = mbrlen( &chars[i], MB_CUR_MAX, NULL );
      printf( "%d bytes in character ", j );
      if( errno == EILSEQ ) {
        printf( " - illegal multibyte character\n" );
      } else {
        if( j == 0 ) {
          k = 0;
        } else if ( j == 1 ) {
          k = chars[i];
        } else if( j == 2 ) {
          k = chars[i]<<8 | chars[i+1];
        }
        printf( "(%#6.4x)\n", k );
      }
    }
  }
```

produces the following:

```
1 bytes in character (0x0020)
1 bytes in character (0x002e)
1 bytes in character (0x0031)
1 bytes in character (0x0041)
2 bytes in character (0x8140)
2 bytes in character (0x8260)
2 bytes in character (0x82a6)
2 bytes in character (0x8342)
1 bytes in character (0x00a1)
1 bytes in character (0x00a6)
1 bytes in character (0x00df)
2 bytes in character (0xe0a1)
0 bytes in character (  0000)
```

Classification: mbrlen is ANSI, _fmbrlen is not ANSI

Systems:
```
mbrlen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbrlen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <wchar.h>
int mbrtowc( wchar_t *pwc, const char *s,
             size_t n, mbstate_t *ps );
int _fmbrtowc( wchar_t __far *pwc, const char __far *s,
               size_t n, mbstate_t __far *ps );
```

Description: If *s* is a null pointer, the `mbrtowc` function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). In this case, the value of the *pwc* argument will be ignored, and the resulting state described will be the initial conversion state.

If *s* is not a null pointer, the `mbrtowc` function determines the number of bytes that are contained in the multibyte character (plus any leading shift sequences) pointed to by *s*, produces the value of the corresponding wide character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described will be the initial conversion state.

The `_fmbrtowc` function is a data model independent form of the `mbrtowc` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The restartable multibyte/wide character conversion functions differ from the corresponding internal-state multibyte character functions (`mblen`, `mbtowc`, and `wctomb`) in that they have an extra argument, *ps*, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead. You are guaranteed that no other function in the library calls these functions with a null pointer for *ps*, thereby ensuring the stability of the state.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

If the encoding is state-dependent, on entry each function takes the described conversion state (either internal or pointed to by *ps*) as current. The conversion state described by the pointed-to object is altered as needed to track the shift state of the associated multibyte character sequence. For encodings without state dependency, the pointer to the `mbstate_t` argument is ignored.

Returns: If *s* is a null pointer, the `mbrtowc` function returns the number of bytes necessary to enter the initial shift state. The value returned will not be greater than that of the `MB_CUR_MAX` macro.

If *s* is not a null pointer, the `mbrtowc` function returns the first of the following that applies:

Value	***Meaning***
0	if the next *n* or fewer bytes form the multibyte character that corresponds to the null wide character.
>0	if the next *n* or fewer bytes form a valid multibyte character; the value returned is the number of bytes that constitute that multibyte character.
-2	if the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed; it is unspecified whether this can occur when the value of *n* is less than that of the `MB_CUR_MAX` macro.
-1	if an encoding error occurs (when the next *n* or fewer bytes do not form a complete and valid multibyte character); the value of the macro `EILSEQ` will be stored in `errno,` but the conversion state will be unchanged.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>


const char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};
```

```
void main()
  {
    int         i, j, k;
    wchar_t     pwc;

    _setmbcp( 932 );
    i = mbrtowc( NULL, NULL, MB_CUR_MAX, NULL );
    printf( "Number of bytes to enter "
            "initial shift state = %d\n", i );
    j = 1;
    for( i = 0; j > 0; i += j ) {
      j = mbrtowc( &pwc, &chars[i], MB_CUR_MAX, NULL );
      printf( "%d bytes in character ", j );
      if( errno == EILSEQ ) {
        printf( " - illegal multibyte character\n" );
      } else {
        if( j == 0 ) {
          k = 0;
        } else if ( j == 1 ) {
          k = chars[i];
        } else if( j == 2 ) {
          k = chars[i]<<8 | chars[i+1];
        }
        printf( "(%#6.4x->%#6.4x)\n", k, pwc );
      }
    }
  }
```

produces the following:

```
Number of bytes to enter initial shift state = 0
1 bytes in character (0x0020->0x0020)
1 bytes in character (0x002e->0x002e)
1 bytes in character (0x0031->0x0031)
1 bytes in character (0x0041->0x0041)
2 bytes in character (0x8140->0x3000)
2 bytes in character (0x8260->0xff21)
2 bytes in character (0x82a6->0x3048)
2 bytes in character (0x8342->0x30a3)
1 bytes in character (0x00a1->0xff61)
1 bytes in character (0x00a6->0xff66)
1 bytes in character (0x00df->0xff9f)
2 bytes in character (0xe0a1->0x720d)
0 bytes in character (  0000->  0000)
```

Classification: mbrtowc is ANSI, _fmbrtowc is not ANSI

Systems: mbrtowc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbrtowc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
#include <mbctype.h> (for manifest constants)
int _mbsbtype( const unsigned char *mbstr, int count );
int _fmbsbtype( const unsigned char __far *mbstr,
                int count );
```

Description: The `_mbsbtype` function determines the type of a byte in a multibyte character string. The function examines only the byte at offset *count* in *mbstr*, ignoring invalid characters before the specified byte

Note: A similar function was called `nthctype` in earlier versions.

Returns: The `_mbsbtype` function returns one of the following values:

_MBC_SINGLE	the character is a valid single-byte character (e.g., 0x20 - 0x7E, 0xA1 - 0xDF in code page 932)
_MBC_LEAD	the character is a valid lead byte character (e.g., 0x81 - 0x9F, 0xE0 - 0xFC in code page 932)
_MBC_TRAIL	the character is a valid trailing byte character (e.g., 0x40 - 0x7E, 0x80 - 0xFC in code page 932)
_MBC_ILLEGAL	the character is an illegal character (e.g., any value except 0x20 - 0x7E, 0xA1 - 0xDF, 0x81 - 0x9F, 0xE0 - 0xFC in code page 932)

See Also: `_getmbcp`, `_ismbcalnum`, `_ismbcalpha`, `_ismbccntrl`, `_ismbcdigit`, `_ismbcgraph`, `_ismbchira`, `_ismbckata`, `_ismbcl0`, `_ismbcl1`, `_ismbcl2`, `_ismbclegal`, `_ismbclower`, `_ismbcprint`, `_ismbcpunct`, `_ismbcspace`, `_ismbcsymbol`, `_ismbcupper`, `_ismbcxdigit`, `_mbbtype`, `_setmbcp`

Example:

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

const char *types[4] = {
    "ILLEGAL",
    "SINGLE",
    "LEAD",
    "TRAIL"
};

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int     i;

    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ )
      printf( "%s\n", types[ 1+_mbsbtype( chars, i ) ] );
  }
```

produces the following:

```
SINGLE
SINGLE
SINGLE
SINGLE
LEAD
TRAIL
LEAD
TRAIL
LEAD
TRAIL
LEAD
TRAIL
SINGLE
SINGLE
SINGLE
LEAD
TRAIL
ILLEGAL
```

Classification: WATCOM

Systems: _mbsbtype - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsbtype - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbsnbcat( unsigned char *dst,
                    const unsigned char *src,
                                 size_t n );
unsigned char __far *_fmbsnbcat( unsigned char __far *dst,
                           const unsigned char __far *src,
                                              size_t n );
```

Description: The `_mbsnbcat` function appends not more than *n* bytes of the string pointed to by *src* to the end of the string pointed to by *dst.* If the byte immediately preceding the null character in *dst* is a lead byte, the initial byte of *src* overwrites this lead byte. Otherwise, the initial byte of *src* overwrites the terminating null character at the end of *dst.* If the last byte to be copied from *src* is a lead byte, the lead byte is not copied and a null character replaces it in *dst.* In any case, a terminating null character is always appended to the result.

The function is a data model independent form of the `_mbsnbcat` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `_mbsnbcat` function returns the value of *dst.*

See Also: `_mbsnbcmp`, `_mbsnbcpy`, `_mbsnbset`, `_mbsnccnt`, `strncat`, `strcat`

Example:

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

const unsigned char str1[] = {
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x00
};

const unsigned char str2[] = {
    0x81,0x40, /* double-byte space */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0x00
};

void main()
  {
    unsigned char   big_string[10];
    int             i;

    _setmbcp( 932 );
    memset( (char *) big_string, 0xee, 10 );
    big_string[9] = 0x00;
    printf( "Length of string = %d\n",
            strlen( (char *) big_string ) );
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );

    _mbsnset( big_string, 0x8145, 5 );
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );

    big_string[0] = 0x00;
    _mbsnbcat( big_string, str1, 3 );
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );

    big_string[2] = 0x84;
    big_string[3] = 0x00;
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );
```

```
        _mbsnbcat( big_string, str2, 5 );
        for( i = 0; i < 10; i++ )
            printf( "%2.2x ", big_string[i] );
        printf( "\n" );

    }
```

produces the following:

```
Length of string = 9
ee ee ee ee ee ee ee ee ee 00
81 45 81 45 81 45 81 45 20 00
81 40 00 00 81 45 81 45 20 00
81 40 84 00 81 45 81 45 20 00
81 40 81 40 82 a6 00 00 20 00
```

Classification: WATCOM

Systems: _mbsnbcat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsnbcat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <mbstring.h>
int _mbsnbcmp( const unsigned char *s1,
               const unsigned char *s2,
               size_t n );
int _fmbsnbcmp( const unsigned char __far *s1,
                const unsigned char __far *s2,
                size_t n );
```

Description: The `_mbsnbcmp` lexicographically compares not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*.

The function is a data model independent form of the `_mbsnbcmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbsnbcmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*. `_mbsnbcmp` is similar to `_mbsncmp,` except that `_mbsnbcmp` compares strings by bytes rather than by characters.

See Also: `_mbsnbcat`, `_mbsnbicmp`, `strncmp`, `strnicmp`

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

const unsigned char str1[] = {
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x00
};

const unsigned char str2[] = {
    0x81,0x40, /* double-byte space */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0x00
};

void main()
  {
    _setmbcp( 932 );
    printf( "%d\n", _mbsnbcmp( str1, str2, 3 ) );
  }
```

produces the following:

```
0
```

Classification: WATCOM

Systems:
```
_mbsnbcmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnbcmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <mbstring.h>
size_t _mbsnbcnt( const unsigned char *string, size_t n );
size_t _fmbsnbcnt( const unsigned char __far *string,
                   size_t n );
#include <tchar.h>
size_t _strncnt( const char *string, size_t n );
size_t _wcsncnt( const wchar_t *string, size_t n ) {
```

Description: The `_mbsnbcnt` function counts the number of bytes in the first *n* multibyte characters of the string *string.*

Note: This function was called `mtob` in earlier versions.

The `_fmbsnbcnt` function is a data model independent form of the `_strncnt` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The header file `<tchar.h>` defines the generic-text routine `_tcsnbcnt.` This macro maps to `_mbsnbcnt` if `_MBCS` has been defined, or to the `_wcsncnt` macro if `_UNICODE` has been defined. Otherwise `_tcsnbcnt` maps to `_strncnt`. `_strncnt` and `_wcsncnt` are single-byte character string and wide-character string versions of `_mbsnbcnt`. The `_strncnt` and `_wcsncnt` macros are provided only for this mapping and should not be used otherwise.

The `_strncnt` function returns the number of characters (i.e., *n*) in the first *n* bytes of the single-byte string *string.* The `_wcsncnt` function returns the number of bytes (i.e., 2 * *n*) in the first *n* wide characters of the wide-character string *string.*

Returns: The `_strncnt` functions return the number of bytes in the string up to the specified number of characters or until a null character is encountered. The null character is not included in the count. If the character preceding the null character was a lead byte, the lead byte is not included in the count.

See Also: `_mbsnbcat`, `_mbsnbcnt`, `_mbsnccnt`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    _setmbcp( 932 );
    printf( "%d bytes found\n",
            _mbsnbcnt( chars, 10 ) );
  }
```

produces the following:

```
14 bytes found
```

Classification: WATCOM

Systems:
```
_mbsnbcnt - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnbcnt - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_strncnt - MACRO
_wcsncnt - MACRO
```

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbsnbcpy( unsigned char *dst,
                    const unsigned char *src,
                    size_t n );
unsigned char __far *_fmbsnbcpy( unsigned char __far *dst,
                           const unsigned char __far *src,
                           size_t n );
```

Description: The `_mbsnbcpy` function copies no more than *n* bytes from the string pointed to by *src* into the array pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly.

If the string pointed to by *src* is shorter than *n* bytes, null characters are appended to the copy in the array pointed to by *dst*, until *n* bytes in all have been written. If the string pointed to by *src* is longer than *n* characters, then the result will not be terminated by a null character.

The function is a data model independent form of the `_mbsnbcpy` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `_mbsnbcpy` function returns the value of *dst*.

See Also: `strcpy`, `strdup`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    unsigned char   chars2[20];
    int             i;

    _setmbcp( 932 );
    _mbsnset( chars2, 0xFF, 20 );
    _mbsnbcpy( chars2, chars, 11 );
    for( i = 0; i < 20; i++ )
        printf( "%2.2x ", chars2[i] );
    printf( "\n" );
    _mbsnbcpy( chars2, chars, 20 );
    for( i = 0; i < 20; i++ )
        printf( "%2.2x ", chars2[i] );
    printf( "\n" );
  }
```

produces the following:

```
20 2e 31 41 81 40 82 60 82 a6 83 ff ff ff ff ff ff ff ff ff
20 2e 31 41 81 40 82 60 82 a6 83 42 a1 a6 df e0 a1 00 00 00
```

Classification: WATCOM

Systems: _mbsnbcpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

```
_fmbsnbcpy - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <mbstring.h>
int _mbsnbicmp( const unsigned char *s1,
               const unsigned char *s2,
               size_t n );
int _fmbsnbicmp( const unsigned char __far *s1,
                const unsigned char __far *s2,
                size_t n );
```

Description: The `_mbsnbicmp` lexicographically compares not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2.* The comparison is insensitive to case.

The function is a data model independent form of the `_mbsnbicmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbsnbicmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2.* `_mbsnbicmp` is similar to `_mbsncmp,` except that `_mbsnbicmp` compares strings by bytes rather than by characters.

See Also: `_mbsnbcat`, `_mbsnbcmp`, `strncmp`, `strnicmp`

Example:
```
#include <stdio.h>
#include <mbctype.h>
#include <mbstring.h>

const unsigned char str1[] = {
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0x79, /* double-byte Z */
    0x00
};

const unsigned char str2[] = {
    0x81,0x40, /* double-byte space */
    0x82,0x81, /* double-byte a */
    0x82,0x9a, /* double-byte z */
    0x00
};

void main()
  {
    _setmbcp( 932 );
    printf( "%d\n", _mbsnbicmp( str1, str2, 5 ) );
  }
```

produces the following:

```
0
```

Classification: WATCOM

Systems:
```
_mbsnbicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnbicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbsnbset( unsigned char *str,
                          unsigned int fill,
                          size_t count );
unsigned char __far *_fmbsnbset( unsigned char __far *str,
                                 unsigned int fill,
                                 size_t count );
```

Description: The `_mbsnbset` function fills the string *str* with the value of the argument *fill*, When the value of *len* is greater than the length of the string, the entire string is filled. Otherwise, that number of characters at the start of the string are set to the fill character.

`_mbsnbset` is similar to `_mbsnset,` except that it fills in *count* bytes rather than *count* characters. If the number of bytes to be filled is odd and *fill* is a double-byte character, the partial byte at the end is filled with an ASCII space character.

The function is a data model independent form of the `_mbsnbset` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The address of the original string *str* is returned.

See Also: `strnset`, `strset`

Example:

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

void main()
  {
    unsigned char   big_string[10];
    int             i;

    _setmbcp( 932 );
    memset( (char *) big_string, 0xee, 10 );
    big_string[9] = 0x00;
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );
    _mbsnbset( big_string, 0x8145, 5 );
    for( i = 0; i < 10; i++ )
        printf( "%2.2x ", big_string[i] );
    printf( "\n" );

  }
```

produces the following:

```
ee ee ee ee ee ee ee ee ee 00
81 45 81 45 20 ee ee ee ee 00
```

Classification: WATCOM

Systems: _mbsnbset - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsnbset - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <mbstring.h>
size_t _mbsnccnt( const unsigned char *string, size_t n );
size_t _fmbsnccnt( const unsigned char __far *string,
                   size_t n );
#include <tchar.h>
size_t _strncnt( const char *string, size_t n );
size_t _wcsncnt( const wchar_t *string, size_t n ) {
```

Description: The `_mbsnccnt` function counts the number of multibyte characters in the first *n* bytes of the string *string.* If `_mbsnccnt` finds a null byte as the second byte of a double-byte character, the first (lead) byte is not included in the count.

Note: This function was called `btom` in earlier versions.

The `_fmbsnccnt` function is a data model independent form of the `_strncnt` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The header file `<tchar.h>` defines the generic-text routine `_tcsnccnt.` This macro maps to `_mbsnccnt` if `_MBCS` has been defined, or to the `_wcsncnt` macro if `_UNICODE` has been defined. Otherwise `_tcsnccnt` maps to `_strncnt`. `_strncnt` and `_wcsncnt` are single-byte character string and wide-character string versions of `_mbsnccnt`. The `_strncnt` and `_wcsncnt` macros are provided only for this mapping and should not be used otherwise.

The `_strncnt` function returns the number of characters (i.e., *n*) in the first *n* bytes of the single-byte string *string.* The `_wcsncnt` function returns the number of bytes (i.e., 2 * *n*) in the first *n* wide characters of the wide-character string *string.*

Returns: `_strncnt` returns the number of characters from the beginning of the string to byte *n.* `_wcsncnt` returns the number of wide characters from the beginning of the string to byte *n.* `_mbsnccnt` returns the number of multibyte characters from the beginning of the string to byte *n.* If these functions find a null character before byte *n,* they return the number of characters before the null character. If the string consists of fewer than *n* characters, these functions return the number of characters in the string.

See Also: `_mbsnbcat`, `_mbsnbcnt`, `_mbsnccnt`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    _setmbcp( 932 );
    printf( "%d characters found\n",
            _mbsnccnt( chars, 10 ) );
  }
```

produces the following:

```
7 characters found
```

Classification: WATCOM

Systems:

```
_mbsnccnt - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnccnt - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_strncnt - MACRO
_wcsncnt - MACRO
```

Synopsis:

```
#include <mbstring.h>
unsigned int _mbsnextc( const unsigned char *string );
unsigned int _fmbsnextc(
                    const unsigned char __far *string );
#include <tchar.h>
unsigned int _strnextc( const char *string );
unsigned int _wcsnextc( const wchar_t *string ) {
```

Description: The `_mbsnextc` function returns the integer value of the next multibyte-character in *string*, without advancing the string pointer. `_mbsnextc` recognizes multibyte character sequences according to the multibyte code page currently in use.

The header file `<tchar.h>` defines the generic-text routine `_tcsnextc.` This macro maps to `_mbsnextc` if `_MBCS` has been defined, or to `_wcsnextc` if `_UNICODE` has been defined. Otherwise `_tcsnextc` maps to `_strnextc`. `_strnextc` and `_wcsnextc` are single-byte character string and wide-character string versions of `_mbsnextc`. `_strnextc` and `_wcsnextc` are provided only for this mapping and should not be used otherwise. `_strnextc` returns the integer value of the next single-byte character in the string. `_wcsnextc` returns the integer value of the next wide character in the string.

Returns: These functions return the integer value of the next character (single-byte, wide, or multibyte) pointed to by *string*.

See Also: `_mbsnextc`, `_strdec`, `_strinc`, `_strninc`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    _setmbcp( 932 );
    printf( "%#6.4x\n", _mbsnextc( &chars[2] ) );
    printf( "%#6.4x\n", _mbsnextc( &chars[4] ) );
    printf( "%#6.4x\n", _mbsnextc( &chars[12] ) );
  }
```

produces the following:

```
0x0031
0x8140
0x00a1
```

Classification: WATCOM

Systems:
```
_mbsnextc - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnextc - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_strnextc - MACRO
_wcsnextc - MACRO
```

Synopsis:

```
#include <wchar.h>
size_t mbsrtowcs( wchar_t *dst,
                  const char **src,
                  size_t len, mbstate_t *ps );
#include <mbstring.h>
size_t _fmbsrtowcs( wchar_t __far *dst,
         const char __far * __far *src,
         size_t len, mbstate_t __far *ps );
```

Description: The `mbsrtowcs` function converts a sequence of multibyte characters that begins in the shift state described by *ps* from the array indirectly pointed to by *src* into a sequence of corresponding wide characters, which, if *dst* is not a null pointer, are then stored into the array pointed to by *dst.* Conversion continues up to and including a terminating null character, but the terminating null wide character will not be stored. Conversion will stop earlier in two cases: when a sequence of bytes is reached that does not form a valid multibyte character, or (if *dst* is not a null pointer) when *len* codes have been stored into the array pointed to by *dst.* Each conversion takes place as if by a call to the `mbrtowc` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* will be assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted. If conversion stopped due to reaching a terminating null character and if *dst* is not a null pointer, the resulting state described will be the initial conversion state.

The `_fmbsrtowcs` function is a data model independent form of the `mbsrtowcs` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The restartable multibyte/wide string conversion functions differ from the corresponding internal-state multibyte string functions (`mbstowcs` and `wcstombs`) in that they have an extra argument, *ps,* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead. You are guaranteed that no other function in the library calls these functions with a null pointer for *ps,* thereby ensuring the stability of the state.

Also unlike their corresponding functions, the conversion source argument, *src,* has a pointer-to-pointer type. When the function is storing conversion results (that is, when *dst* is not a null pointer), the pointer object pointed to by this argument will be updated to reflect the amount of the source processed by that invocation.

If the encoding is state-dependent, on entry each function takes the described conversion state (either internal or pointed to by *ps*) as current and then, if the destination pointer, *dst,* is not a null pointer, the conversion state described by the pointed-to object is altered as needed to track the shift state of the associated multibyte character sequence. For encodings without state dependency, the pointer to the `mbstate_t` argument is ignored.

Returns: If the input string does not begin with a valid multibyte character, an encoding error occurs: The `mbsrtowcs` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)-1,` but the conversion state will be unchanged. Otherwise, it returns the number of multibyte characters successfully converted, which is the same as the number of array elements modified when *dst* is not a null pointer.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>

const char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};
```

```
void main()
  {
    int         i;
    size_t      elements;
    char        *src;
    wchar_t     wc[50];
    mbstate_t   pstate;


    _setmbcp( 932 );
    src = chars;
    elements = mbsrtowcs( wc, &src, 50, &pstate );
    if( errno == EILSEQ ) {
      printf( "Error in multibyte character string\n" );
    } else {
      for( i = 0; i < elements; i++ ) {
        printf( "%#6.4x\n", wc[i] );
      }
    }
  }
```

produces the following:

```
0x0020
0x002e
0x0031
0x0041
0x3000
0xff21
0x3048
0x30a3
0xff61
0xff66
0xff9f
0x720d
```

Classification: mbsrtowcs is ANSI, _fmbsrtowcs is not ANSI

Systems:

```
mbsrtowcs - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsrtowcs - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs, const char *s, size_t n );
#include <mbstring.h>
size_t _fmbstowcs( const wchar_t __far *pwcs,
                   char __far *s,
                   size_t n );
```

Description: The `mbstowcs` function converts a sequence of multibyte characters pointed to by *s* into their corresponding wide character codes and stores not more than *n* codes into the array pointed to by *pwcs*. The `mbstowcs` function does not convert any multibyte characters beyond the null character. At most *n* elements of the array pointed to by *pwcs* will be modified.

The `_fmbstowcs` function is a data model independent form of the `mbstowcs` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: If an invalid multibyte character is encountered, the `mbstowcs` function returns `(size_t)-1.` Otherwise, the `mbstowcs` function returns the number of array elements modified, not including the terminating zero code if present.

See Also: `mblen`, `mbtowc`, `wctomb`, `wcstombs`

Example:

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

void main()
  {
    char    *wc = "string";
    wchar_t wbuffer[50];
    int     i, len;

    len = mbstowcs( wbuffer, wc, 50 );
    if( len != -1 ) {
      wbuffer[len] = '\0';
      printf( "%s(%d)\n", wc, len );
      for( i = 0; i < len; i++ )
        printf( "/%4.4x", wbuffer[i] );
      printf( "\n" );
    }
  }
```

produces the following:

```
string(6)
/0073/0074/0072/0069/006e/0067
```

Classification: mbstowcs is ANSI, _fmbstowcs is not ANSI

Systems:
```
mbstowcs - All, Netware
_fmbstowcs - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <mbstring.h>
int _mbterm( const unsigned char *ch );
int _fmbterm( const unsigned char __far *ch );
```

Description: The `_mbterm` function determines if the next multibyte character in the string pointed to by *ch* is a null character or a valid lead byte followed by a null character.

The function is a data model independent form of the `_mbterm` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbterm` function returns 1 if the multibyte character pointed to by *ch* is a null character. The `_mbterm` function returns 2 if the multibyte character pointed to by *ch* is a valid lead byte character followed by a null character. Otherwise, the `_mbterm` function returns 0.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x00  /* invalid double-byte */
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int     i, j, k;

    _setmbcp( 932 );
    k = 0;
    for( i = 0; i < SIZE; i++ ) {
      printf( "0x%2.2x %d\n", chars[i],
              _mbterm( &chars[i] ) );
    }
  }
```

produces the following:

```
0x20 0
0x2e 0
0x31 0
0x41 0
0x81 0
0x40 0
0x82 2
0x00 1
```

Classification: WATCOM

Systems:
```
_mbterm - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbterm - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc, const char *s, size_t n );
#include <mbstring.h>
int _fmbtowc( wchar_t __far *pwc,
              const char __far *s,
              size_t n );
```

Description: The `mbtowc` function converts a single multibyte character pointed to by *s* into the wide character code that corresponds to that multibyte character. The code for the null character is zero. If the multibyte character is valid and *pwc* is not a NULL pointer, the code is stored in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

The `mbtowc` function does not examine more than `MB_CUR_MAX` bytes.

The `_fmbtowc` function is a data model independent form of the `mbtowc` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: If *s* is a NULL pointer, the `mbtowc` function returns zero if multibyte character encodings are not state dependent, and non-zero otherwise. If *s* is not a NULL pointer, the `mbtowc` function returns:

Value	***Meaning***
0	if *s* points to the null character
len	the number of bytes that comprise the multibyte character (if the next *n* or fewer bytes form a valid multibyte character)
-1	if the next *n* bytes do not form a valid multibyte character

See Also: `mblen`, `wctomb`, `mbstowcs`, `wcstombs`

Example:

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

void main()
  {
    char    *wc = "string";
    wchar_t wbuffer[10];
    int     i, len;
    _setmbcp( 932 );
    printf( "Character encodings are %sstate dependent\n",
            ( mbtowc( wbuffer, NULL, 0 ) )
            ? "" : "not " );

    len = mbtowc( wbuffer, wc, MB_CUR_MAX );
    wbuffer[len] = '\0';
    printf( "%s(%d)\n", wc, len );
    for( i = 0; i < len; i++ )
        printf( "/%4.4x", wbuffer[i] );
    printf( "\n" );
  }
```

produces the following:

```
Character encodings are not state dependent
string(1)
/0073
```

Classification: mbtowc is ANSI, _fmbtowc is not ANSI

Systems:
```
mbtowc - All, Netware
_fmbtowc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <mbstring.h>
unsigned char *_mbvtop( unsigned int ch,
                        unsigned char *addr );
unsigned char __far *_fmbvtop( unsigned int ch,
                        unsigned char __far *addr );
```

Description: The `_mbvtop` function stores the multibyte character *ch* into the string pointed to by *addr*.

The function is a data model independent form of the `_mbvtop` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `_mbvtop` function returns the value of the argument *addr*.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:

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


void main()
  {
    unsigned char string[10];
    unsigned char *p;
    int           i;

    _setmbcp( 932 );
    p = string;
    _mbvtop( '.', p );
    p++;
    _mbvtop( '1', p );
    p++;
    _mbvtop( 'A', p );
    p++;
    _mbvtop( 0x8140, p );
    p += 2;
    _mbvtop( 0x8260, p );
    p += 2;
    _mbvtop( 0x82A6, p );
    p += 2;
    _mbvtop( '\0', p );

    for( i = 0; i < 10; i++ )
      printf( "%2.2x ", string[i] );
    printf( "\n" );
  }
```

produces the following:

```
2e 31 41 81 40 82 60 82 a6 00
```

Classification: WATCOM

Systems: _mbvtop - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbvtop - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <malloc.h>
size_t _memavl( void );
```

Description: The `_memavl` function returns the number of bytes of memory available for dynamic memory allocation in the near heap (the default data segment). In the tiny, small and medium memory models, the default data segment is only extended as needed to satisfy requests for memory allocation. Therefore, you will need to call `_nheapgrow` in these memory models before calling `_memavl` in order to get a meaningful result.

The number returned by `_memavl` may not represent a single contiguous block of memory. Use the `_memmax` function to find the largest contiguous block of memory that can be allocated.

Returns: The `_memavl` function returns the number of bytes of memory available for dynamic memory allocation in the near heap (the default data segment).

See Also: `calloc` Functions, `_freect`, `_memmax`, `_heapgrow` Functions, `malloc` Functions, `realloc` Functions

Example:
```
#include <stdio.h>
#include <malloc.h>

void main()
  {
    char *p;
    char *fmt = "Memory available = %u\n";

    printf( fmt, _memavl() );
    _nheapgrow();
    printf( fmt, _memavl() );
    p = (char *) malloc( 2000 );
    printf( fmt, _memavl() );
  }
```

produces the following:

```
Memory available = 0
Memory available = 62732
Memory available = 60730
```

Classification: WATCOM

Systems: All

Synopsis:

```
#include <string.h>
void *memccpy( void *dest, const void *src,
               int c, size_t cnt );
void __far *_fmemccpy( void __far *dest,
                       const void __far *src,
                       int c, size_t cnt );
```

Description: The `memccpy` function copies bytes from *src* to *dest* up to and including the first occurrence of the character *c* or until *cnt* bytes have been copied, whichever comes first.

The `_fmemccpy` function is a data model independent form of the `memccpy` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `memccpy` function returns a pointer to the byte in *dest* following the character *c* if one is found and copied, otherwise it returns NULL.

See Also: `memcpy`, `memmove`, `memset`

Example:

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

char *msg = "This is the string: not copied";

void main()
  {
    auto char buffer[80];

    memset( buffer, '\0', 80 );
    memccpy( buffer, msg, ':', 80 );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
This is the string:
```

Classification: WATCOM

Systems:

```
memccpy - All, Netware
_fmemccpy - All
```

Synopsis:

```
#include <string.h>
void *memchr( const void *buf, int ch, size_t length );
void __far *_fmemchr( const void __far *buf,
                      int ch,
                      size_t length );
```

Description: The `memchr` function locates the first occurrence of *ch* (converted to an unsigned char) in the first *length* characters of the object pointed to by *buf.*

The `_fmemchr` function is a data model independent form of the `memchr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `memchr` function returns a pointer to the located character, or `NULL` if the character does not occur in the object.

See Also: `memcmp`, `memcpy`, `memicmp`, `memset`

Example:

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

void main()
  {
    char buffer[80];
    char *where;

    strcpy( buffer, "video x-rays" );
    where = (char *) memchr( buffer, 'x', 6 );
    if( where == NULL )
      printf( "'x' not found\n" );
    else
      printf( "%s\n", where );
    where = (char *) memchr( buffer, 'r', 9 );
    if( where == NULL )
      printf( "'r' not found\n" );
    else
      printf( "%s\n", where );
  }
```

Classification: memchr is ANSI, _fmemchr is not ANSI

Systems:

```
memchr - All, Netware
_fmemchr - All
```

Synopsis:

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

Description: The `memcmp` function compares the first *length* characters of the object pointed to by *s1* to the object pointed to by *s2*.

The `_fmemcmp` function is a data model independent form of the `memcmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `memcmp` function returns an integer less than, equal to, or greater than zero, indicating that the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

See Also: `memchr`, `memcpy`, `memicmp`, `memset`

Example:

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

void main()
  {
    auto char buffer[80];

    strcpy( buffer, "world" );
    if( memcmp( buffer, "Hello ", 6 ) < 0 ) {
      printf( "Less than\n" );
    }
  }
```

Classification: memcmp is ANSI, _fmemcmp is not ANSI

Systems:

```
memcmp - All, Netware
_fmemcmp - All
```

Synopsis:
```
#include <string.h>
void *memcpy( void *dst,
              const void *src,
              size_t length );
void __far *_fmemcpy( void __far *dst,
                      const void __far *src,
                      size_t length );
```

Description: The memcpy function copies *length* characters from the buffer pointed to by *src* into the buffer pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly. See the memmove function if you wish to copy objects that overlap.

The _fmemcpy function is a data model independent form of the memcpy function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The original value of *dst* is returned.

See Also: memchr, memcmp, memicmp, memmove, memset

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

void main()
  {
    auto char buffer[80];

    memcpy( buffer, "Hello", 5 );
    buffer[5] = ’\0’;
    printf( "%s\n", buffer );
  }
```

Classification: memcpy is ANSI, _fmemcpy is not ANSI

Systems:
```
memcpy - All, Netware
_fmemcpy - All
```

Synopsis:
```
#include <string.h>
int memicmp( const void *s1,
             const void *s2,
             size_t length );
int _memicmp( const void *s1,
              const void *s2,
              size_t length );
int _fmemicmp( const void __far *s1,
               const void __far *s2,
               size_t length );
```

Description: The memicmp function compares, with case insensitivity (upper- and lowercase characters are equivalent), the first *length* characters of the object pointed to by *s1* to the object pointed to by *s2*.

The _fmemicmp function is a data model independent form of the memicmp function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The _memicmp function is identical to memicmp. Use _memicmp for ANSI/ISO naming conventions.

Returns: The memicmp function returns an integer less than, equal to, or greater than zero, indicating that the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

See Also: memchr, memcmp, memcpy, memset

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

void main()
  {
    char buffer[80];

    if( memicmp( buffer, "Hello", 5 ) < 0 ) {
      printf( "Less than\n" );
    }
  }
```

Classification: WATCOM

_memicmp conforms to ANSI/ISO naming conventions

Systems:

```
memicmp - All, Netware
_memicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmemicmp - All
```

Synopsis:
```
#include <malloc.h>
size_t _memmax( void );
```

Description: The `_memmax` function returns the size of the largest contiguous block of memory available for dynamic memory allocation in the near heap (the default data segment). In the tiny, small and medium memory models, the default data segment is only extended as needed to satisfy requests for memory allocation. Therefore, you will need to call `_nheapgrow` in these memory models before calling `_memmax` in order to get a meaningful result.

Returns: The `_memmax` function returns the size of the largest contiguous block of memory available for dynamic memory allocation in the near heap. If 0 is returned, then there is no more memory available in the near heap.

See Also: `calloc`, `_freect`, `_memavl`, `_heapgrow`, `malloc`

Example:
```
#include <stdio.h>
#include <malloc.h>

void main()
  {
    char *p;
    size_t size;

    size = _memmax();
    printf( "Maximum memory available is %u\n", size );
    _nheapgrow();
    size = _memmax();
    printf( "Maximum memory available is %u\n", size );
    p = (char *) _nmalloc( size );
    size = _memmax();
    printf( "Maximum memory available is %u\n", size );
  }
```

produces the following:

```
Maximum memory available is 0
Maximum memory available is 62700
Maximum memory available is 0
```

Classification: WATCOM

Systems: All

Synopsis:

```
#include <string.h>
void *memmove( void *dst,
               const void *src,
               size_t length );
void __far *_fmemmove( void __far *dst,
                       const void __far *src,
                       size_t length );
```

Description: The memmove function copies *length* characters from the buffer pointed to by *src* to the buffer pointed to by *dst.* Copying of overlapping objects will take place properly. See the memcpy function to copy objects that do not overlap.

The _fmemmove function is a data model independent form of the memmove function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The memmove function returns *dst.*

See Also: memchr, memcmp, memcpy, memicmp, memset

Example:

```
#include <string.h>

void main()
  {
    char buffer[80];

    memmove( buffer+1, buffer, 79 );
    buffer[0] = '*';
  }
```

Classification: memmove is ANSI, _fmemmove is not ANSI

Systems:

```
memmove - All, Netware
_fmemmove - All
```

Synopsis:
```
#include <mmintrin.h>
void  _m_empty(void);
```

Description: The `_m_empty` function empties the multimedia state. The values in the Multimedia Tag Word (TW) are set to empty (i.e., all ones). This will indicate that no Multimedia registers are in use.

This function is useful for applications that mix floating-point (FP) instructions with multimedia instructions. Intel maps the multimedia registers onto the floating-point registers. For this reason, you are discouraged from intermixing MM code and FP code. The recommended way to write an application with FP instructions and MM instructions is:

- Split the FP code and MM code into two separate instruction streams such that each stream contains only instructions of one type.
- Do not rely on the contents of FP/MM registers across transitions from one stream to the other.
- Leave the MM state empty at the end of an MM stream using the `_m_empty` function.
- Similarly, leave the FP stack empty at the end of an FP stream.

Returns: The `_m_empty` function does not return a value.

See Also: `_m_from_int`, `_m_to_int`, `_m_packsswb`, `_m_paddb`, `_m_pand`, `_m_pcmpeqb`, `_m_pmaddwd`, `_m_psllw`, `_m_psraw`, `_m_psrlw`, `_m_psubb`, `_m_punpckhbw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

long featureflags(void);

#pragma aux featureflags = \
    ".586"          \
    "mov eax,1"     \
    "CPUID"         \
    "mov eax,edx"   \
    modify [eax ebx ecx edx]

#define MM_EXTENSION 0x00800000
```

```
main()
  {
    if( featureflags() & MM_EXTENSION ) {
    /*
        sequence of code that uses Multimedia functions
        .
        .
        .
    */

        _m_empty();
    }

    /*
        sequence of code that uses floating-point
        .
        .
        .
    */
  }
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <string.h>
void *memset( void *dst, int c, size_t length );
void __far *_fmemset( void __far *dst, int c,
                      size_t length );
```

Description: The `memset` function fills the first *length* characters of the object pointed to by *dst* with the value *c.*

The `_fmemset` function is a data model independent form of the `memset` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

Returns: The `memset` function returns the pointer *dst.*

See Also: `memchr`, `memcmp`, `memcpy`, `memicmp`, `memmove`

Example:

```
#include <string.h>

void main()
  {
    char buffer[80];

    memset( buffer, '=', 80 );
  }
```

Classification: memset is ANSI, _fmemset is not ANSI

Systems:

```
memset - All, Netware
_fmemset - All
```

Synopsis:

```
#include <mmintrin.h>
__m64 _m_from_int(int i);
```

Description: The `_m_from_int` function forms a 64-bit MM value from an unsigned 32-bit integer value.

Returns: The 64-bit result of loading MM0 with an unsigned 32-bit integer value is returned.

See Also: `_m_empty`, `_m_to_int`, `_m_packsswb`, `_m_paddb`, `_m_pand`, `_m_empty`, `_m_pcmpeqb`, `_m_pmaddwd`, `_m_psllw`, `_m_psraw`, `_m_psrlw`, `_m_empty`, `_m_psubb`, `_m_punpckhbw`

Example:

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

__m64   a;

int     k = 0xF1F2F3F4;

void main()
  {
    a = _m_from_int( k );
    printf( "int=%8.8lx m=%8.8lx%8.8lx\n",
        k, a._32[1], a._32[0] );

  }
```

produces the following:

```
int=f1f2f3f4 m=00000000f1f2f3f4
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <stdlib.h>
#define min(a,b)  (((a) < (b)) ? (a) : (b))
```

Description: The `min` macro will evaluate to be the lesser of two values. It is implemented as follows.

```
#define min(a,b)  (((a) < (b)) ? (a) : (b))
```

Returns: The `min` macro will evaluate to the smaller of the two values passed.

See Also: `max`

Example:

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

void main()
  {
    int a;

    /*
     * The following line will set the variable "a" to 1
     * since 10 is greater than 1.
     */
    a = min( 1, 10 );
    printf( "The value is: %d\n", a );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <sys\types.h>
#include <direct.h>
int mkdir( const char *path );
int _mkdir( const char *path );
int _wmkdir( const wchar_t *path );
```

Description: The `mkdir` function creates a new subdirectory with name *path.* The *path* can be either relative to the current working directory or it can be an absolute path name.

The `_mkdir` function is identical to `mkdir`. Use `_mkdir` for ANSI/ISO naming conventions.

The `_wmkdir` function is identical to `mkdir` except that it accepts a wide-character string argument.

Returns: The `mkdir` function returns zero if successful, and a non-zero value otherwise.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EACCES	Search permission is denied for a component of *path* or write permission is denied on the parent directory of the directory to be created.
ENOENT	The specified *path* does not exist or *path* is an empty string.

See Also: `chdir`, `chmod`, `getcwd`, `rmdir`, `stat`, `umask`

Example: To make a new directory called `\watcom` on drive `C:`

```
#include <sys\types.h>
#include <direct.h>

void main()
  {
    mkdir( "c:\\watcom" );
  }
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: mkdir is POSIX 1003.1, _mkdir is not POSIX, _wmkdir is not POSIX _mkdir conforms to ANSI/ISO naming conventions

Systems:

```
mkdir - All, Netware
_mkdir - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wmkdir - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <i86.h>
void __far *MK_FP( unsigned int segment,
                   unsigned int offset );
```

Description: The `MK_FP` macro can be used to obtain the far pointer value given by the *segment* segment value and the *offset* offset value. These values may be obtained by using the `FP_SEG` and `FP_OFF` macros.

Returns: The macro returns a far pointer.

See Also: `FP_OFF`, `FP_SEG`, `segread`

Example:

```
#include <i86.h>
#include <stdio.h>

void main()
  {
    unsigned short __far *bios_prtr_port_1;

    bios_prtr_port_1 =
            (unsigned short __far *) MK_FP( 0x40, 0x8 );
    printf( "Port address is %x\n", *bios_prtr_port_1 );
  }
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <io.h>
char *_mktemp( char *template );
#include <wchar.h>
wchar_t *_wmktemp( wchar_t *template );
```

Description: The `_mktemp` function creates a unique filename by modifying the *template* argument. `_mktemp` automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use by the run-time system.

The `_wmktemp` function is a wide-character version of `_mktemp` that operates with wide-character strings.

The string *template* has the form `baseXXXXXX` where `base` is the fixed part of the generated filename and `XXXXXX` is the variable part of the generated filename. Each of the 6 X's is a placeholder for a character supplied by `_mktemp`. Each placeholder character in *template* must be an uppercase "X". `_mktemp` preserves `base` and replaces the first of the 6 trailing X's with a lowercase alphabetic character (a-z). `_mktemp` replaces the following 5 trailing X's with a five-digit value this value is a unique number identifying the calling process or thread.

`_mktemp` checks to see if a file with the generated name already exists and if so selects another letter, in succession, from "a" to "z" until it finds a file that doesn't exist. If it is unsuccessful at finding a name for a file that does not already exist, `_mktemp` returns NULL. At most, 26 unique file names can be returned to the calling process or thread.

Returns: The `_mktemp` function returns a pointer to the modified *template*. The `_mktemp` function returns NULL if *template* is badly formed or no more unique names can be created from the given template.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `fopen`, `freopen`, `_tempnam`, `tmpfile`, `tmpnam`

Example:

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

#define TMPLTE "_tXXXXXX"

void main()
  {
    char name[sizeof(TMPLTE)];
    char *mknm;
    int i;
    FILE *fp;

    for( i = 0; i < 30; i++ ) {
      strcpy( name, TMPLTE );
      mknm = _mktemp( name );
      if( mknm == NULL )
        printf( "Name is badly formed\n" );
      else {
        printf( "Name is %s\n", mknm );
        fp = fopen( mknm, "w" );
        if( fp != NULL ) {
          fprintf( fp, "Name is %s\n", mknm );
          fclose( fp );
        }
      }
    }
  }
```

Classification: WATCOM

Systems: `_mktemp - Win32`
`_wmktemp - Win32`

Synopsis:

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

struct  tm {
  int tm_sec;   /* seconds after the minute -- [0,61] */
  int tm_min;   /* minutes after the hour   -- [0,59] */
  int tm_hour;  /* hours after midnight     -- [0,23] */
  int tm_mday;  /* day of the month         -- [1,31] */
  int tm_mon;   /* months since January     -- [0,11] */
  int tm_year;  /* years since 1900                   */
  int tm_wday;  /* days since Sunday        -- [0,6]  */
  int tm_yday;  /* days since January 1     -- [0,365]*/
  int tm_isdst; /* Daylight Savings Time flag */
};
```

Description: The `mktime` function converts the local time information in the structure pointed to by *timeptr* into a calendar time (Coordinated Universal Time) with the same encoding used by the `time` function. The original values of the fields `tm_sec`, `tm_min`, `tm_hour`, `tm_mday`, and `tm_mon` are not restricted to ranges described for `struct tm.` If these fields are not in their proper ranges, they are adjusted so that they are in the proper ranges. Values for the fields `tm_wday` and `tm_yday` are computed after all the other fields have been adjusted.

If the original value of `tm_isdst` is negative, this field is computed also. Otherwise, a value of 0 is treated as "daylight savings time is not in effect" and a positive value is treated as "daylight savings time is in effect".

Whenever `mktime` is called, the `tzset` function is also called.

Returns: The `mktime` function returns the converted calendar time.

See Also: `asctime`, `clock`, `ctime`, `difftime`, `gmtime`, `localtime`, `strftime`, `time`, `tzset`

Example:

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

static const char *week_day[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

```
void main()
  {
    struct tm new_year;

    new_year.tm_year  = 2001 - 1900;
    new_year.tm_mon   = 0;
    new_year.tm_mday  = 1;
    new_year.tm_hour  = 0;
    new_year.tm_min   = 0;
    new_year.tm_sec   = 0;
    new_year.tm_isdst = 0;
    mktime( &new_year );
    printf( "The next century begins on a %s\n",
             week_day[ new_year.tm_wday ] );
  }
```

produces the following:

```
The next century begins on a Monday
```

Classification: ANSI

Systems: All, Netware

Synopsis:

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

Description: The `modf` function breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` in the object pointed to by *iptr*.

Returns: The `modf` function returns the signed fractional part of *value*.

See Also: `frexp`, `ldexp`

Example:

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

void main()
  {
    double integral_value, fractional_part;

    fractional_part = modf( 4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );
    fractional_part = modf( -4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );
  }
```

produces the following:

```
0.500000 4.000000
-0.500000 -4.000000
```

Classification: ANSI

Systems: Math

Synopsis:
```
#include <string.h>
void movedata( unsigned int src_segment,
               unsigned int src_offset,
               unsigned int tgt_segment,
               unsigned int tgt_offset,
               size_t length );
```

Description: The `movedata` function copies *length* bytes from the far pointer calculated as `(src_segment:src_offset)` to a target location determined as a far pointer `(tgt_segment:tgt_offset)`.

Overlapping data may not be correctly copied. When the source and target areas may overlap, copy the areas one character at a time.

The function is useful to move data when the near address(es) of the source and/or target areas are not known.

Returns: No value is returned.

See Also: `FP_SEG`, `FP_OFF`, `memcpy`, `segread`

Example:
```
#include <stdio.h>
#include <string.h>
#include <dos.h>

void main()
  {
    char buffer[14] = {
        '*', 0x17, 'H', 0x17, 'e', 0x17, 'l', 0x17,
        'l', 0x17, 'o', 0x17, '*', 0x17 };

    movedata( FP_SEG( buffer ),
              FP_OFF( buffer ),
              0xB800,
              0x0720,
              14 );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <graph.h>
struct xycoord _FAR _moveto( short x, short y );

struct _wxycoord _FAR _moveto_w( double x, double y );
```

Description: The `_moveto` functions set the current output position for graphics. The `_moveto` function uses the view coordinate system. The `_moveto_w` function uses the window coordinate system.

The current output position is set to be the point at the coordinates `(x,y).` Nothing is drawn by the function. The `_lineto` function uses the current output position as the starting point when a line is drawn.

Note that the output position for graphics output differs from that for text output. The output position for text output can be set by use of the `_settextposition` function.

Returns: The `_moveto` functions return the previous value of the output position for graphics.

See Also: `_getcurrentposition`, `_lineto`, `_settextposition`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _moveto( 100, 100 );
    _lineto( 540, 100 );
    _lineto( 320, 380 );
    _lineto( 100, 100 );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems:

```
_moveto - DOS, QNX
_moveto_w - DOS, QNX
```

Synopsis:
```
#include <mmintrin.h>
__m64 _m_packssdw(__m64 *m1, __m64 *m2);
```

Description: Convert signed packed double-words into signed packed words by packing (with signed saturation) the low-order words of the signed double-word elements from *m1* and *m2* into the respective signed words of the result. If the signed values in the word elements of *m1* and *m2* are smaller than 0x8000, the result elements are clamped to 0x8000. If the signed values in the word elements of *m1* and *m2* are larger than 0x7fff, the result elements are clamped to 0x7fff.

```
        m2                  m1
---------------------   ---------------------
| w3 : w2 | w1 : w0 |   | w3 : w2 | w1 : w0 |
---------------------   ---------------------
     |         |              |         |
     `--------.`---.     .---'.--------'
              |    |     |    |
              V    V     V    V
            ---------------------
            | w3 | w2 | w1 | w0 |
            ---------------------
                    result
```

Returns: The result of packing, with signed saturation, 32-bit signed double-words into 16-bit signed words is returned.

See Also: `_m_empty`, `_m_packsswb`, `_m_packuswb`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"
#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0000567800001234 };
__m64   c = { 0xfffffffe00010101 };
```

```
void main()
  {
    a = _m_packssdw( b, c );
    printf( "m2="AS_DWORDS" "
            "m1="AS_DWORDS"\n"
            "mm="AS_WORDS"\n",
        c._32[1], c._32[0],
        b._32[1], b._32[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m2=fffffffe 00010101 m1=00005678 00001234
mm=fffe 7fff 5678 1234
```

Classification: Intel

Systems: MACRO

_m_packsswb

Synopsis:

```
#include <mmintrin.h>
__m64 _m_packsswb(__m64 *m1, __m64 *m2);
```

Description: Convert signed packed words into signed packed bytes by packing (with signed saturation) the low-order bytes of the signed word elements from *m1* and *m2* into the respective signed bytes of the result. If the signed values in the word elements of *m1* and *m2* are smaller than 0x80, the result elements are clamped to 0x80. If the signed values in the word elements of *m1* and *m2* are larger than 0x7f, the result elements are clamped to 0x7f.

```
            m2                         m1
-------------------------  -------------------------
|b7 b6|b5 b4|b3 b2|b1 b0|  |b7 b6|b5 b4|b3 b2|b1 b0|
-------------------------  -------------------------
   |     |     |     |        |     |     |     |
   |     |     |     `--.  .--'     |     |     |
   |     |     `-----.  |  |  .-----'     |     |
   |     `--------.  |  |  |  |  .--------'     |
   `-----------.  |  |  |  |  |  |  .-----------'
               |  |  |  |  |  |  |  |
               V  V  V  V  V  V  V  V
              -------------------------
              |b7|b6|b5|b4|b3|b2|b1|b0|
              -------------------------
                    result
```

Returns: The result of packing, with signed saturation, 16-bit signed words into 8-bit signed bytes is returned.

See Also: `_m_empty`, `_m_packssdw`, `_m_packuswb`

Example:

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

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"
#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_packsswb( b, c );
    printf( "m2="AS_WORDS" "
            "m1="AS_WORDS"\n"
            "mm="AS_BYTES"\n",
        c._16[3], c._16[2], c._16[1], c._16[0],
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m2=ff7f ff80 0080 007f m1=0004 0003 0002 0001
mm=80 80 7f 7f 04 03 02 01
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_packuswb(__m64 *m1, __m64 *m2);
```

Description: Convert signed packed words into unsigned packed bytes by packing (with unsigned saturation) the low-order bytes of the signed word elements from *m1* and *m2* into the respective unsigned bytes of the result. If the signed values in the word elements of *m1* and *m2* are too large to be represented in an unsigned byte, the result elements are clamped to 0xff.

```
                m2                         m1
    -------------------------  -------------------------
    |b7 b6|b5 b4|b3 b2|b1 b0|  |b7 b6|b5 b4|b3 b2|b1 b0|
    -------------------------  -------------------------
       |     |     |     |        |     |     |     |
       |     |     |     ‘--.  .--’     |     |     |
       |     |     ‘-----.  |  |  .-----’     |     |
       |     ‘--------.  |  |  |  |  .--------’     |
       ‘-----------.  |  |  |  |  |  |  .-----------’
                   |  |  |  |  |  |  |  |
                   V  V  V  V  V  V  V  V
                  -------------------------
                  |b7|b6|b5|b4|b3|b2|b1|b0|
                  -------------------------
                        result
```

Returns: The result of packing, with unsigned saturation, 16-bit signed words into 8-bit unsigned bytes is returned.

See Also: `_m_empty`, `_m_packssdw`, `_m_packsswb`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"
#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_packuswb( b, c );
    printf( "m2="AS_WORDS" "
            "m1="AS_WORDS"\n"
            "mm="AS_BYTES"\n",
        c._16[3], c._16[2], c._16[1], c._16[0],
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m2=ff7f ff80 0080 007f m1=0004 0003 0002 0001
mm=00 00 80 7f 04 03 02 01
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddb(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 8-bit bytes of *m2* are added to the respective signed or unsigned 8-bit bytes of *m1* and the result is stored in memory. If any result element does not fit into 8 bits (overflow), the lower 8 bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of adding the packed bytes of two 64-bit multimedia values is returned.

See Also: _m_empty, _m_paddd, _m_paddsb, _m_paddsw, _m_paddusb, _m_paddusw, _m_paddw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_paddb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=01 23 45 67 89 ab cd ef
m2=fe dc ba 98 76 54 32 10
mm=ff ff ff ff ff ff ff ff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddd(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 32-bit double-words of *m2* are added to the respective signed or unsigned 32-bit double-words of *m1* and the result is stored in memory. If any result element does not fit into 32 bits (overflow), the lower 32-bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of adding the packed double-words of two 64-bit multimedia values is returned.

See Also: `_m_empty`, `_m_paddb`, `_m_paddsb`, `_m_paddsw`, `_m_paddusb`, `_m_paddusw`, `_m_paddw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_paddd( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c._32[1], c._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=01234567 89abcdef
m2=fedcba98 76543210
mm=ffffffff ffffffff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddsb(__m64 *m1, __m64 *m2);
```

Description: The signed 8-bit bytes of *m2* are added to the respective signed 8-bit bytes of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of a signed byte. In the case where a result is a byte larger than 0x7f (overflow), it is clamped to 0x7f. In the case where a result is a byte smaller than 0x80 (underflow), it is clamped to 0x80.

Returns: The result of adding the packed signed bytes, with saturation, of two 64-bit multimedia values is returned.

See Also: `_m_empty`, `_m_paddb`, `_m_paddd`, `_m_paddsw`, `_m_paddusb`, `_m_paddusw`, `_m_paddw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_paddsb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=8a ac ce ef 02 24 46 68
m2=76 54 32 11 fe dc ba 98
mm=00 00 00 00 00 00 00 00
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddsw(__m64 *m1, __m64 *m2);
```

Description: The signed 16-bit words of *m2* are added to the respective signed 16-bit words of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of a signed word. In the case where a result is a word larger than 0x7fff (overflow), it is clamped to 0x7fff. In the case where a result is a word smaller than 0x8000 (underflow), it is clamped to 0x8000.

Returns: The result of adding the packed signed words, with saturation, of two 64-bit multimedia values is returned.

See Also: _m_empty, _m_paddb, _m_paddd, _m_paddsb, _m_paddusb, _m_paddusw, _m_paddw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_paddsw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=8aac ceef 0224 4668
m2=7654 3211 fedc ba98
mm=0100 0100 0100 0100
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddusb(__m64 *m1, __m64 *m2);
```

Description: The unsigned 8-bit bytes of *m2* are added to the respective unsigned 8-bit bytes of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of an unsigned byte. In the case where a result is a byte larger than 0xff (overflow), it is clamped to 0xff.

Returns: The result of adding the packed unsigned bytes, with saturation, of two 64-bit multimedia values is returned.

See Also: `_m_empty`, `_m_paddb`, `_m_paddd`, `_m_paddsb`, `_m_paddsw`, `_m_paddusw`, `_m_paddw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_paddusb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=8a ac ce ef 02 24 46 68
m2=76 54 32 11 fe dc ba 98
mm=ff ff ff ff ff ff ff ff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddusw(__m64 *m1, __m64 *m2);
```

Description: The unsigned 16-bit words of *m2* are added to the respective unsigned 16-bit words of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of an unsigned word. In the case where a result is a word larger than 0xffff (overflow), it is clamped to 0xffff.

Returns: The result of adding the packed unsigned words, with saturation, of two 64-bit multimedia values is returned.

See Also: `_m_empty`, `_m_paddb`, `_m_paddd`, `_m_paddsb`, `_m_paddsw`, `_m_paddusb`, `_m_paddw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_paddusw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=8aac ceef 0224 4668
m2=7654 3211 fedc ba98
mm=ffff ffff ffff ffff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_paddw(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 16-bit words of *m2* are added to the respective signed or unsigned 16-bit words of *m1* and the result is stored in memory. If any result element does not fit into 16 bits (overflow), the lower 16 bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of adding the packed words of two 64-bit multimedia values is returned.

See Also: `_m_empty`, `_m_paddb`, `_m_paddd`, `_m_paddsb`, `_m_paddsw`, `_m_paddusb`, `_m_paddusw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_paddw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=0123 4567 89ab cdef
m2=fedc ba98 7654 3210
mm=ffff ffff ffff ffff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pand(__m64 *m1, __m64 *m2);
```

Description: A bit-wise logical AND is performed between 64-bit multimedia operands *m1* and *m2* and the result is stored in memory.

Returns: The bit-wise logical AND of two 64-bit values is returned.

See Also: `_m_empty`, `_m_pandn`, `_m_por`, `_m_pxor`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_pand( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=0123456789abcdef
m2=fedcba9876543210
mm=0000000000000000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pandn(__m64 *m1, __m64 *m2);
```

Description: A bit-wise logical AND is performed on the logical inversion of 64-bit multimedia operand *m1* and 64-bit multimedia operand *m2* and the result is stored in memory.

Returns: The bit-wise logical AND of an inverted 64-bit value and a non-inverted value is returned.

See Also: _m_empty, _m_pand, _m_por, _m_pxor

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_pandn( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=0123456789abcdef
m2=fedcba9876543210
mm=fedcba9876543210
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpeqb(__m64 *m1, __m64 *m2);
```

Description: If the respective bytes of *m1* are equal to the respective bytes of *m2*, the respective bytes of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the packed bytes of two 64-bit multimedia values is returned as a sequence of bytes (0xff for equal, 0x00 for not equal).

See Also: _m_empty, _m_pcmpeqd, _m_pcmpeqw, _m_pcmpgtb, _m_pcmpgtd, _m_pcmpgtw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };

void main()
  {
    a = _m_pcmpeqb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=00 04 00 03 00 02 00 01
m2=ff 7f ff 80 00 80 00 7f
mm=00 00 00 00 ff 00 ff 00
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpeqd(__m64 *m1, __m64 *m2);
```

Description: If the respective double-words of *m1* are equal to the respective double-words of *m2*, the respective double-words of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the 32-bit packed double-words of two 64-bit multimedia values is returned as a sequence of double-words (0xffffffff for equal, 0x00000000 for not equal).

See Also: `_m_empty`, `_m_pcmpeqb`, `_m_pcmpeqw`, `_m_pcmpgtb`, `_m_pcmpgtd`, `_m_pcmpgtw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0x000400030002007f };

void main()
  {
    a = _m_pcmpeqd( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c._32[1], c._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=00040003 00020001
m2=00040003 0002007f
mm=ffffffff 00000000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpeqw(__m64 *m1, __m64 *m2);
```

Description: If the respective words of *m1* are equal to the respective words of *m2*, the respective words of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the packed words of two 64-bit multimedia values is returned as a sequence of words (0xffff for equal, 0x0000 for not equal).

See Also: `_m_empty`, `_m_pcmpeqb`, `_m_pcmpeqd`, `_m_pcmpgtb`, `_m_pcmpgtd`, `_m_pcmpgtw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0x0004ff8000800001 };

void main()
  {
    a = _m_pcmpeqw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=0004 0003 0002 0001
m2=0004 ff80 0080 0001
mm=ffff 0000 0000 ffff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpgtb(__m64 *m1, __m64 *m2);
```

Description: If the respective signed bytes of *m1* are greater than the respective signed bytes of *m2*, the respective bytes of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the packed signed bytes of two 64-bit multimedia values is returned as a sequence of bytes (0xff for greater than, 0x00 for not greater than).

See Also: _m_empty, _m_pcmpeqb, _m_pcmpeqd, _m_pcmpeqw, _m_pcmpgtd, _m_pcmpgtw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };

void main()
  {
    a = _m_pcmpgtb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=00 04 00 03 00 02 00 01
m2=ff 7f ff 80 00 80 00 7f
mm=ff 00 ff ff 00 ff 00 00
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpgtd(__m64 *m1, __m64 *m2);
```

Description: If the respective signed double-words of *m1* are greater than the respective signed double-words of *m2*, the respective double-words of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the 32-bit packed signed double-words of two 64-bit multimedia values is returned as a sequence of double-words (0xffffffff for greater than, 0x00000000 for not greater than).

See Also: `_m_empty`, `_m_pcmpeqb`, `_m_pcmpeqd`, `_m_pcmpeqw`, `_m_pcmpgtb`, `_m_pcmpgtw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000400020001 };
__m64   c = { 0x000400030080007f };

void main()
  {
    a = _m_pcmpgtd( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c._32[1], c._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=00040004 00020001
m2=00040003 0080007f
mm=ffffffff 00000000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pcmpgtw(__m64 *m1, __m64 *m2);
```

Description: If the respective signed words of *m1* are greater than the respective signed words of *m2*, the respective words of the result are set to all ones, otherwise they are set to all zeros.

Returns: The result of comparing the 16-bit packed signed words of two 64-bit multimedia values is returned as a sequence of words (0xffff for greater than, 0x0000 for not greater than).

See Also: `_m_empty`, `_m_pcmpeqb`, `_m_pcmpeqd`, `_m_pcmpeqw`, `_m_pcmpgtb`, `_m_pcmpgtd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0005000300020001 };
__m64   c = { 0x0004ff8000800001 };

void main()
  {
    a = _m_pcmpgtw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=0005 0003 0002 0001
m2=0004 ff80 0080 0001
mm=ffff ffff 0000 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pmaddwd(__m64 *m1, __m64 *m2);
```

Description: The signed 16-bit words of *m1* are multiplied with the respective signed 16-bit words of *m2*. The 32-bit intermediate results are summed by pairs producing two 32-bit integers.

```
    MM[63-32] = M1[63-48] x M2[63-48]
              + M1[47-32] x M2[47-32]
    MM[31-0]  = M1[31-16] x M2[31-16]
              + M1[15-0]  x M2[15-0]
```

In cases which overflow, the results are truncated. These two integers are packed into their respective elements of the result.

Returns: The result of multiplying the packed signed 16-bit words of two 64-bit multimedia values and adding the 32-bit results pairwise is returned as packed double-words.

See Also: `_m_empty`, `_m_pmulhw`, `_m_pmullw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0000006000123456 };
__m64   c = { 0x0000000200010020 };

void main()
  {
    a = _m_pmaddwd( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_DWORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=0000 0060 0012 3456
m2=0000 0002 0001 0020
mm=000000c0 00068ad2
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pmulhw(__m64 *m1, __m64 *m2);
```

Description: The signed 16-bit words of *m1* are multiplied with the respective signed 16-bit words of *m2*. The high-order 16-bits of each result are placed in the respective elements of the result.

Returns: The packed 16-bit words in *m1* are multiplied with the packed 16-bit words in *m2* and the high-order 16-bits of the results are returned.

See Also: `_m_empty`, `_m_pmaddwd`, `_m_pmullw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x4000006000123456 };
__m64   c = { 0x0008000210000020 };

void main()
  {
    a = _m_pmulhw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=4000 0060 0012 3456
m2=0008 0002 1000 0020
mm=0002 0000 0001 0006
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pmullw(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 16-bit words of *m1* are multiplied with the respective signed or unsigned 16-bit words of *m2*. The low-order 16-bits of each result are placed in the respective elements of the result.

Returns: The packed 16-bit words in *m1* are multiplied with the packed 16-bit words in *m2* and the low-order 16-bits of the results are returned.

See Also: `_m_empty`, `_m_pmaddwd`, `_m_pmulhw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x4000006000123456 };
__m64   c = { 0x0008000210000020 };

void main()
  {
    a = _m_pmullw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=4000 0060 0012 3456
m2=0008 0002 1000 0020
mm=0000 00c0 2000 8ac0
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_por(__m64 *m1, __m64 *m2);
```

Description: A bit-wise logical OR is performed between 64-bit multimedia operands *m1* and *m2* and the result is stored in memory.

Returns: The bit-wise logical OR of two 64-bit values is returned.

See Also: `_m_empty`, `_m_pand`, `_m_pandn`, `_m_pxor`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_por( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=0123456789abcdef
m2=fedcba9876543210
mm=ffffffffffffffff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pslld(__m64 *m, __m64 *count);
```

Description: The 32-bit double-words in *m* are each independently shifted to the left by the scalar shift count in *count*. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all zeros.

Returns: Shift left each 32-bit double-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: _m_empty, _m_pslldi, _m_psllq, _m_psllqi, _m_psllw, _m_psllwi

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_pslld( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c,
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=3f048003 00020001
m2=0000000000000002
mm=fc12000c 00080004
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pslldi(__m64 *m, int count);
```

Description: The 32-bit double-words in *m* are each independently shifted to the left by the scalar shift count in *count*. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all zeros.

Returns: Shift left each 32-bit double-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_pslld`, `_m_psllq`, `_m_psllqi`, `_m_psllw`, `_m_psllwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_pslldi( b, 2 );
    printf( "m ="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m =3f048003 00020001
mm=fc12000c 00080004
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <mmintrin.h>
__m64 _m_psllq(__m64 *m, __m64 *count);
```

Description: The 64-bit quad-word in *m* is shifted to the left by the scalar shift count in *count*. The low-order bits are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 63 yield all zeros.

Returns: Shift left the 64-bit quad-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_pslld`, `_m_pslldi`, `_m_psllqi`, `_m_psllw`, `_m_psllwi`

Example:

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

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psllq( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=3f04800300020001
m2=0000000000000002
mm=fc12000c00080004
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psllqi(__m64 *m, int count);
```

Description: The 64-bit quad-word in *m* is shifted to the left by the scalar shift count in *count*. The low-order bits are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 63 yield all zeros.

Returns: Shift left the 64-bit quad-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_pslld`, `_m_pslldi`, `_m_psllq`, `_m_psllw`, `_m_psllwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_psllqi( b, 2 );
    printf( "m ="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, a );
  }
```

produces the following:

```
m =3f04800300020001
mm=fc12000c00080004
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psllw(__m64 *m, __m64 *count);
```

Description: The 16-bit words in *m* are each independently shifted to the left by the scalar shift count in *count*. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all zeros.

Returns: Shift left each 16-bit word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_pslld`, `_m_pslldi`, `_m_psllq`, `_m_psllqi`, `_m_psllwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psllw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c,
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=3f04 8003 0002 0001
m2=0000000000000002
mm=fc10 000c 0008 0004
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <mmintrin.h>
__m64 _m_psllwi(__m64 *m, int count);
```

Description: The 16-bit words in *m* are each independently shifted to the left by the scalar shift count in *count*. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all zeros.

Returns: Shift left each 16-bit word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_pslld`, `_m_pslldi`, `_m_psllq`, `_m_psllqi`, `_m_psllw`

Example:

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

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_psllwi( b, 2 );
    printf( "m ="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m =3f04 8003 0002 0001
mm=fc10 000c 0008 0004
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrad(__m64 *m, __m64 *count);
```

Description: The 32-bit signed double-words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with the initial value of the sign bit of each element. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all ones or zeros depending on the initial value of the sign bit.

Returns: Shift right each 32-bit double-word in *m* by an amount specified in *count* while shifting in sign bits.

See Also: `_m_empty`, `_m_psradi`, `_m_psraw`, `_m_psrawi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psrad( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c,
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=3f048003 00020001
m2=0000000000000002
mm=0fc12000 00008000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psradi(__m64 *m, int count);
```

Description: The 32-bit signed double-words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with the initial value of the sign bit of each element. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all ones or zeros depending on the initial value of the sign bit.

Returns: Shift right each 32-bit double-word in *m* by an amount specified in *count* while shifting in sign bits.

See Also: `_m_empty`, `_m_psrad`, `_m_psraw`, `_m_psrawi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_psradi( b, 2 );
    printf( "m ="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m =3f048003 00020001
mm=0fc12000 00008000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psraw(__m64 *m, __m64 *count);
```

Description: The 16-bit signed words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with the initial value of the sign bit of each element. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all ones or zeros depending on the initial value of the sign bit.

Returns: Shift right each 16-bit word in *m* by an amount specified in *count* while shifting in sign bits.

See Also: `_m_empty`, `_m_psrad`, `_m_psradi`, `_m_psrawi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300040001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psraw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c,
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=3f04 8003 0004 0001
m2=0000000000000002
mm=0fc1 e000 0001 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrawi(__m64 *m, int count);
```

Description: The 16-bit signed words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with the initial value of the sign bit of each element. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all ones or zeros depending on the initial value of the sign bit.

Returns: Shift right each 16-bit word in *m* by an amount specified in *count* while shifting in sign bits.

See Also: `_m_empty`, `_m_psrad`, `_m_psradi`, `_m_psraw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x3f04800300040001 };

void main()
  {
    a = _m_psrawi( b, 2 );
    printf( "m ="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m =3f04 8003 0004 0001
mm=0fc1 e000 0001 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrld(__m64 *m, __m64 *count);
```

Description: The 32-bit double-words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all zeros.

Returns: Shift right each 32-bit double-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrldi`, `_m_psrlq`, `_m_psrlqi`, `_m_psrlw`, `_m_psrlwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psrld( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c,
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=3f048003 00020001
m2=0000000000000002
mm=0fc12000 00008000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrldi(__m64 *m, int count);
```

Description: The 32-bit double-words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 31 yield all zeros.

Returns: Shift right each 32-bit double-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrld`, `_m_psrlq`, `_m_psrlqi`, `_m_psrlw`, `_m_psrlwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_psrldi( b, 2 );
    printf( "m ="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m =3f048003 00020001
mm=0fc12000 00008000
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <mmintrin.h>
__m64 _m_psrlq(__m64 *m, __m64 *count);
```

Description: The 64-bit quad-word in *m* is shifted to the right by the scalar shift count in *count*. The high-order bits are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 63 yield all zeros.

Returns: Shift right the 64-bit quad-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrld`, `_m_psrldi`, `_m_psrlqi`, `_m_psrlw`, `_m_psrlwi`

Example:

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

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psrlq( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=3f04800300020001
m2=0000000000000002
mm=0fc12000c0008000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrlqi(__m64 *m, int count);
```

Description: The 64-bit quad-word in *m* is shifted to the right by the scalar shift count in *count*. The high-order bits are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 63 yield all zeros.

Returns: Shift right the 64-bit quad-word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrld`, `_m_psrldi`, `_m_psrlq`, `_m_psrlw`, `_m_psrlwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300020001 };

void main()
  {
    a = _m_psrlqi( b, 2 );
    printf( "m ="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, a );
  }
```

produces the following:

```
m =3f04800300020001
mm=0fc12000c0008000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrlw(__m64 *m, __m64 *count);
```

Description: The 16-bit words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all zeros.

Returns: Shift right each 16-bit word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrld`, `_m_psrldi`, `_m_psrlq`, `_m_psrlqi`, `_m_psrlwi`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"
#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x3f04800300040001 };
__m64   c = { 0x0000000000000002 };

void main()
  {
    a = _m_psrlw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c,
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=3f04 8003 0004 0001
m2=0000000000000002
mm=0fc1 2000 0001 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psrlwi(__m64 *m, int count);
```

Description: The 16-bit words in *m* are each independently shifted to the right by the scalar shift count in *count*. The high-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 yield all zeros.

Returns: Shift right each 16-bit word in *m* by an amount specified in *count* while shifting in zeros.

See Also: `_m_empty`, `_m_psrld`, `_m_psrldi`, `_m_psrlq`, `_m_psrlqi`, `_m_psrlw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x3f04800300040001 };

void main()
  {
    a = _m_psrlwi( b, 2 );
    printf( "m ="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m =3f04 8003 0004 0001
mm=0fc1 2000 0001 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubb(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 8-bit bytes of *m2* are subtracted from the respective signed or unsigned 8-bit bytes of *m1* and the result is stored in memory. If any result element does not fit into 8 bits (underflow or overflow), the lower 8 bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of subtracting the packed bytes of one 64-bit multimedia value from another is returned.

See Also: `_m_empty`, `_m_psubd`, `_m_psubsb`, `_m_psubsw`, `_m_psubusb`, `_m_psubusw`, `_m_psubw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_psubb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=01 23 45 67 89 ab cd ef
m2=fe dc ba 98 76 54 32 10
mm=03 47 8b cf 13 57 9b df
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubd(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 32-bit double-words of *m2* are subtracted from the respective signed or unsigned 32-bit double-words of *m1* and the result is stored in memory. If any result element does not fit into 32 bits (underflow or overflow), the lower 32-bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of subtracting one set of packed double-words from a second set of packed double-words is returned.

See Also: `_m_empty`, `_m_psubb`, `_m_psubsb`, `_m_psubsw`, `_m_psubusb`, `_m_psubusw`, `_m_psubw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_psubd( b, c );
    printf( "m1="AS_DWORDS"\n"
            "m2="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        b._32[1], b._32[0],
        c._32[1], c._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m1=01234567 89abcdef
m2=fedcba98 76543210
mm=02468acf 13579bdf
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubsb(__m64 *m1, __m64 *m2);
```

Description: The signed 8-bit bytes of *m2* are subtracted from the respective signed 8-bit bytes of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of a signed byte. In the case where a result is a byte larger than 0x7f (overflow), it is clamped to 0x7f. In the case where a result is a byte smaller than 0x80 (underflow), it is clamped to 0x80.

Returns: The result of subtracting the packed signed bytes, with saturation, of one 64-bit multimedia value from a second multimedia value is returned.

See Also: `_m_empty`, `_m_psubb`, `_m_psubd`, `_m_psubsw`, `_m_psubusb`, `_m_psubusw`, `_m_psubw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_psubsb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=8a ac ce ef 02 24 46 68
m2=76 54 32 11 fe dc ba 98
mm=80 80 9c de 04 48 7f 7f
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <mmintrin.h>
__m64 _m_psubsw(__m64 *m1, __m64 *m2);
```

Description: The signed 16-bit words of *m2* are subtracted from the respective signed 16-bit words of *m1* and the result is stored in memory. Saturation occurs when a result exceeds the range of a signed word. In the case where a result is a word larger than 0x7fff (overflow), it is clamped to 0x7fff. In the case where a result is a word smaller than 0x8000 (underflow), it is clamped to 0x8000.

Returns: The result of subtracting the packed signed words, with saturation, of one 64-bit multimedia value from a second multimedia value is returned.

See Also: `_m_empty`, `_m_psubb`, `_m_psubd`, `_m_psubsb`, `_m_psubusb`, `_m_psubusw`, `_m_psubw`

Example:

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

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_psubsw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=8aac ceef 0224 4668
m2=7654 3211 fedc ba98
mm=8000 9cde 0348 7fff
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubusb(__m64 *m1, __m64 *m2);
```

Description: The unsigned 8-bit bytes of *m2* are subtracted from the respective unsigned 8-bit bytes of *m1* and the result is stored in memory. Saturation occurs when a result is less than zero. If a result is less than zero, it is clamped to 0xff.

Returns: The result of subtracting the packed unsigned bytes, with saturation, of one 64-bit multimedia value from a second multimedia value is returned.

See Also: `_m_empty`, `_m_psubb`, `_m_psubd`, `_m_psubsb`, `_m_psubsw`, `_m_psubusw`, `_m_psubw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_psubusb( b, c );
    printf( "m1="AS_BYTES"\n"
            "m2="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m1=8a ac ce ef 02 24 46 68
m2=76 54 32 11 fe dc ba 98
mm=14 58 9c de 00 00 00 00
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubusw(__m64 *m1, __m64 *m2);
```

Description: The unsigned 16-bit words of *m2* are subtracted from the respective unsigned 16-bit words of *m1* and the result is stored in memory. Saturation occurs when a result is less than zero. If a result is less than zero, it is clamped to 0xffff.

Returns: The result of subtracting the packed unsigned words, with saturation, of one 64-bit multimedia value from a second multimedia value is returned.

See Also: _m_empty, _m_psubb, _m_psubd, _m_psubsb, _m_psubsw, _m_psubusb, _m_psubw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x8aacceef02244668 };
__m64   c = { 0x76543211fedcba98 };

void main()
  {
    a = _m_psubusw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=8aac ceef 0224 4668
m2=7654 3211 fedc ba98
mm=1458 9cde 0000 0000
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_psubw(__m64 *m1, __m64 *m2);
```

Description: The signed or unsigned 16-bit words of *m2* are subtracted from the respective signed or unsigned 16-bit words of *m1* and the result is stored in memory. If any result element does not fit into 16 bits (underflow or overflow), the lower 16 bits of the result elements are stored (i.e., truncation takes place).

Returns: The result of subtracting the packed words of two 64-bit multimedia values is returned.

See Also: _m_empty, _m_psubb, _m_psubd, _m_psubsb, _m_psubsw, _m_psubusb, _m_psubusw

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_psubw( b, c );
    printf( "m1="AS_WORDS"\n"
            "m2="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        b._16[3], b._16[2], b._16[1], b._16[0],
        c._16[3], c._16[2], c._16[1], c._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m1=0123 4567 89ab cdef
m2=fedc ba98 7654 3210
mm=0247 8acf 1357 9bdf
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpckhbw(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpckhbw` function performs an interleaved unpack of the high-order data elements of *m1* and *m2*. It ignores the low-order bytes. When unpacking from a memory operand, the full 64-bit operand is accessed from memory but only the high-order 32 bits are utilized. By choosing *m1* or *m2* to be zero, an unpacking of byte elements into word elements is performed.

```
             m2                          m1
-------------------------   -------------------------
|b7|b6|b5|b4|b3|b2|b1|b0|   |b7|b6|b5|b4|b3|b2|b1|b0|
-------------------------   -------------------------
 |  |  |  |                   |  |  |  |
 V  V  V  V                   V  V  V  V
 b7 b5 b3 b1                  b6 b4 b2 b0

              -------------------------
              |b7|b6|b5|b4|b3|b2|b1|b0|
              -------------------------
                    result
```

Returns: The result of the interleaved unpacking of the high-order bytes of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhdq`, `_m_punpckhwd`, `_m_punpcklbw`, `_m_punpckldq`, `_m_punpcklwd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_punpckhbw( b, c );
    printf( "m2="AS_BYTES" "
            "m1="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m2=ff 7f ff 80 00 80 00 7f m1=00 04 00 03 00 02 00 01
mm=ff 00 7f 04 ff 00 80 03
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpckhdq(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpckhdq` function performs an interleaved unpack of the high-order data elements of *m1* and *m2*. It ignores the low-order double-words. When unpacking from a memory operand, the full 64-bit operand is accessed from memory but only the high-order 32 bits are utilized.

```
                m2                         m1
    -------------------------  -------------------------
    |     d1    |     d0    |  |     d1    |     d0    |
    -------------------------  -------------------------
          |                          |
          v                          v
          d1                         d0

                  -------------------------
                  |     d1    |     d0    |
                  -------------------------
                        result
```

Returns: The result of the interleaved unpacking of the high-order double-words of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhbw`, `_m_punpckhwd`, `_m_punpcklbw`, `_m_punpckldq`, `_m_punpcklwd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_punpckhdq( b, c );
    printf( "m2="AS_DWORDS" "
            "m1="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        c._32[1], c._32[0],
        b._32[1], b._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m2=ff7fff80 0080007f m1=00040003 00020001
mm=ff7fff80 00040003
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpckhwd(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpckhwd` function performs an interleaved unpack of the high-order data elements of *m1* and *m2*. It ignores the low-order words. When unpacking from a memory operand, the full 64-bit operand is accessed from memory but only the high-order 32 bits are utilized. By choosing *m1* or *m2* to be zero, an unpacking of word elements into double-word elements is performed.

```
             m2                          m1
  -------------------------   -------------------------
  |  w3 |  w2 |  w1 |  w0 |   |  w3 |  w2 |  w1 |  w0 |
  -------------------------   -------------------------
     |     |                        |    |
     V     V                        V    V
     w3    w1                      w2    w0

                 -------------------------
                 |  w3 |  w2 |  w1 |  w0 |
                 -------------------------
                       result
```

Returns: The result of the interleaved unpacking of the high-order words of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhbw`, `_m_punpckhdq`, `_m_punpcklbw`, `_m_punpckldq`, `_m_punpcklwd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_punpckhwd( b, c );
    printf( "m2="AS_WORDS" "
            "m1="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        c._16[3], c._16[2], c._16[1], c._16[0],
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m2=ff7f ff80 0080 007f m1=0004 0003 0002 0001
mm=ff7f 0004 ff80 0003
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpcklbw(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpcklbw` function performs an interleaved unpack of the low-order data elements of *m1* and *m2*. It ignores the high-order bytes. When unpacking from a memory operand, 32 bits are accessed and all are utilized by the instruction. By choosing *m1* or *m2* to be zero, an unpacking of byte elements into word elements is performed.

```
            m2                            m1
-------------------------  -------------------------
|           |b3|b2|b1|b0|  |b7|b6|b5|b4|b3|b2|b1|b0|
-------------------------  -------------------------
             |  |  |  |                 |  |  |  |
             V  V  V  V                 V  V  V  V
             b7 b5 b3 b1                b6 b4 b2 b0

              -------------------------
              |b7|b6|b5|b4|b3|b2|b1|b0|
              -------------------------
                    result
```

Returns: The result of the interleaved unpacking of the low-order bytes of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhbw`, `_m_punpckhdq`, `_m_punpckhwd`, `_m_punpckldq`, `_m_punpcklwd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_BYTES "%2.2x %2.2x %2.2x %2.2x " \
                 "%2.2x %2.2x %2.2x %2.2x"

__m64   a;
__m64   b = { 0x000200013478bcf0 };
__m64   c = { 0x0080007f12569ade };
```

```
void main()
  {
    a = _m_punpcklbw( b, c );
    printf( "m2="AS_BYTES" "
            "m1="AS_BYTES"\n"
            "mm="AS_BYTES"\n",
        c._8[7], c._8[6], c._8[5], c._8[4],
        c._8[3], c._8[2], c._8[1], c._8[0],
        b._8[7], b._8[6], b._8[5], b._8[4],
        b._8[3], b._8[2], b._8[1], b._8[0],
        a._8[7], a._8[6], a._8[5], a._8[4],
        a._8[3], a._8[2], a._8[1], a._8[0] );
  }
```

produces the following:

```
m2=00 80 00 7f 12 56 9a de m1=00 02 00 01 34 78 bc f0
mm=12 34 56 78 9a bc de f0
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpckldq(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpckldq` function performs an interleaved unpack of the low-order data elements of *m1* and *m2*. It ignores the high-order double-words. When unpacking from a memory operand, 32 bits are accessed and all are utilized by the instruction.

```
            m2                           m1
-------------------------   -------------------------
|     d1     |     d0     |   |     d1     |     d0     |
-------------------------   -------------------------
                  |                            |
                  V                            V
                  d1                           d0

              -------------------------
              |     d1     |     d0     |
              -------------------------
                    result
```

Returns: The result of the interleaved unpacking of the low-order double-words of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhbw`, `_m_punpckhdq`, `_m_punpckhwd`, `_m_punpcklbw`, `_m_punpcklwd`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_DWORDS "%8.8lx %8.8lx"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };

void main()
  {
    a = _m_punpckldq( b, c );
    printf( "m2="AS_DWORDS" "
            "m1="AS_DWORDS"\n"
            "mm="AS_DWORDS"\n",
        c._32[1], c._32[0],
        b._32[1], b._32[0],
        a._32[1], a._32[0] );
  }
```

produces the following:

```
m2=ff7fff80 0080007f m1=00040003 00020001
mm=0080007f 00020001
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_punpcklwd(__m64 *m1, __m64 *m2);
```

Description: The `_m_punpcklwd` function performs an interleaved unpack of the low-order data elements of *m1* and *m2*. It ignores the high-order words. When unpacking from a memory operand, 32 bits are accessed and all are utilized by the instruction. By choosing *m1* or *m2* to be zero, an unpacking of word elements into double-word elements is performed.

```
             m2                           m1
-------------------------  -------------------------
|  w3 |  w2 |  w1 |  w0 |  |  w3 |  w2 |  w1 |  w0 |
-------------------------  -------------------------
               |     |                      |    |
               v     v                      v    v
               w3    w1                    w2    w0

              -------------------------
              |  w3 |  w2 |  w1 |  w0 |
              -------------------------
                    result
```

Returns: The result of the interleaved unpacking of the low-order words of two multimedia values is returned.

See Also: `_m_empty`, `_m_punpckhbw`, `_m_punpckhdq`, `_m_punpckhwd`, `_m_punpcklbw`, `_m_punpckldq`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_WORDS "%4.4x %4.4x %4.4x %4.4x"

__m64   a;
__m64   b = { 0x0004000300020001 };
__m64   c = { 0xff7fff800080007f };
```

```
void main()
  {
    a = _m_punpcklwd( b, c );
    printf( "m2="AS_WORDS" "
            "m1="AS_WORDS"\n"
            "mm="AS_WORDS"\n",
        c._16[3], c._16[2], c._16[1], c._16[0],
        b._16[3], b._16[2], b._16[1], b._16[0],
        a._16[3], a._16[2], a._16[1], a._16[0] );
  }
```

produces the following:

```
m2=ff7f ff80 0080 007f m1=0004 0003 0002 0001
mm=0080 0002 007f 0001
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <mmintrin.h>
__m64 _m_pxor(__m64 *m1, __m64 *m2);
```

Description: A bit-wise logical XOR is performed between 64-bit multimedia operands *m1* and *m2* and the result is stored in memory.

Returns: The bit-wise logical exclusive OR of two 64-bit values is returned.

See Also: _m_empty, _m_pand, _m_pandn, _m_por

Example:
```
#include <stdio.h>
#include <mmintrin.h>

#define AS_QWORD "%16.16Lx"

__m64   a;
__m64   b = { 0x0123456789abcdef };
__m64   c = { 0xfedcba9876543210 };

void main()
  {
    a = _m_pxor( b, c );
    printf( "m1="AS_QWORD"\n"
            "m2="AS_QWORD"\n"
            "mm="AS_QWORD"\n",
            b, c, a );
  }
```

produces the following:

```
m1=0123456789abcdef
m2=fedcba9876543210
mm=ffffffffffffffff
```

Classification: Intel

Systems: MACRO

Synopsis:

```
#include <malloc.h>
size_t _msize( void *buffer );
size_t _bmsize( __segment seg, void __based(void) *buffer );
size_t _fmsize( void __far *buffer );
size_t _nmsize( void __near *buffer );
```

Description: The `_msize` functions return the size of the memory block pointed to by *buffer* that was allocated by a call to the appropriate version of the `calloc`, `malloc`, or `realloc` functions.

You must use the correct `_msize` function as listed below depending on which heap the memory block belongs to.

Function	***Heap***
_msize	Depends on data model of the program
_bmsize	Based heap specified by *seg* value
_fmsize	Far heap (outside the default data segment)
_nmsize	Near heap (inside the default data segment)

In small data models (small and medium memory models), `_msize` maps to `_nmsize.` In large data models (compact, large and huge memory models), `_msize` maps to `_fmsize.`

Returns: The `_msize` functions return the size of the memory block pointed to by *buffer*.

See Also: `calloc` Functions, `_expand` Functions, `free` Functions, `halloc`, `hfree`, `malloc` Functions, `realloc` Functions, `sbrk`

Example:

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

void main()
  {
    void *buffer;

    buffer = malloc( 999 );
    printf( "Size of block is %u bytes\n",
                _msize( buffer ) );
  }
```

produces the following:

```
Size of block is 1000 bytes
```

Classification: WATCOM

Systems:
```
_msize - All, Netware
_bmsize - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_fmsize - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_nmsize - DOS, Windows, Win386, Win32, QNX, OS/2 1.x, OS/2
1.x(MT), OS/2-32
```

Synopsis:
```
#include <mmintrin.h>
int   _m_to_int(__m64 *__m);
```

Description: The `_m_to_int` function returns the low-order 32 bits of a multimedia value.

Returns: The low-order 32 bits of a multimedia value are fetched and returned as the result.

See Also: `_m_empty`, `_m_from_int`, `_m_packsswb`, `_m_paddb`, `_m_pand`, `_m_empty`, `_m_pcmpeqb`, `_m_pmaddwd`, `_m_psllw`, `_m_psraw`, `_m_psrlw`, `_m_empty`, `_m_psubb`, `_m_punpckhbw`

Example:
```
#include <stdio.h>
#include <mmintrin.h>

__m64   b = { 0x0123456789abcdef };

int     j;

void main()
  {
    j = _m_to_int( b );
    printf( "m=%16.16Lx int=%8.8lx\n",
            b, j );
  }
```

produces the following:

```
m=0123456789abcdef int=89abcdef
```

Classification: Intel

Systems: MACRO

Synopsis:
```
#include <i86.h>
void nosound( void );
```

Description: The `nosound` function turns off the PC's speaker.

Returns: The `nosound` function has no return value.

See Also: `delay`, `sound`

Example:
```
#include <i86.h>

void main()
  {
    sound( 200 );
    delay( 500 );  /* delay for 1/2 second */
    nosound();
  }
```

Classification: Intel

Systems: DOS, Windows, Win386, QNX

Synopsis:
```
#include <stddef.h>
size_t offsetof( composite, name );
```

Description: The `offsetof` macro returns the offset of the element *name* within the `struct` or `union` *composite*. This provides a portable method to determine the offset.

Returns: The `offsetof` function returns the offset of *name*.

Example:
```
#include <stdio.h>
#include <stddef.h>

struct new_def
{  char *first;
   char second[10];
   int third;
};

void main()
  {
    printf( "first:%d second:%d third:%d\n",
        offsetof( struct new_def, first ),
        offsetof( struct new_def, second ),
        offsetof( struct new_def, third ) );
  }
```

produces the following:

In a small data model, the following would result:

```
first:0 second:2 third:12
```

In a large data model, the following would result:

```
first:0 second:4 third:14
```

Classification: ANSI

Systems: MACRO

Synopsis:
```
#include <stdlib.h>
onexit_t onexit( onexit_t func );
```

Description: The `onexit` function is passed the address of function *func* to be called when the program terminates normally. Successive calls to `onexit` create a list of functions that will be executed on a "last-in, first-out" basis. No more than 32 functions can be registered with the `onexit` function.

The functions have no parameters and do not return values.

NOTE: The `onexit` function is not an ANSI function. The ANSI standard function `atexit` does the same thing that `onexit` does and should be used instead of `onexit` where ANSI portability is concerned.

Returns: The `onexit` function returns *func* if the registration succeeds, NULL if it fails.

See Also: `abort`, `atexit`, `exit`, `_exit`

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

void main()
  {
    extern void func1(void), func2(void), func3(void);

    onexit( func1 );
    onexit( func2 );
    onexit( func3 );
    printf( "Do this first.\n" );
  }

void func1(void) { printf( "last.\n" ); }
void func2(void) { printf( "this " ); }
void func3(void) { printf( "Do " ); }
```

produces the following:

```
Do this first.
Do this last.
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <sys\types.h>
#include <sys\stat.h>
#include <fcntl.h>
int open( const char *path, int access, ... );
int _open( const char *path, int access, ... );
int _wopen( const wchar_t *path, int access, ... );
```

Description: The `open` function opens a file at the operating system level. The name of the file to be opened is given by *path.* The file will be accessed according to the access mode specified by *access.* The optional argument is the file permissions to be used when the `O_CREAT` flag is on in the *access* mode.

The `_open` function is identical to `open`. Use `_open` for ANSI/ISO naming conventions.

The `_wopen` function is identical to `open` except that it accepts a wide character string argument for *path.*

The access mode is established by a combination of the bits defined in the `<fcntl.h>` header file. The following bits may be set:

Mode	***Meaning***
O_RDONLY	permit the file to be only read.
O_WRONLY	permit the file to be only written.
O_RDWR	permit the file to be both read and written.
O_APPEND	causes each record that is written to be written at the end of the file.
O_CREAT	has no effect when the file indicated by *filename* already exists; otherwise, the file is created;
O_TRUNC	causes the file to be truncated to contain no data when the file exists; has no effect when the file does not exist.
O_BINARY	causes the file to be opened in binary mode which means that data will be transmitted to and from the file unchanged.
O_TEXT	causes the file to be opened in text mode which means that carriage-return characters are written before any linefeed character

	that is written and causes carriage-return characters to be removed when encountered during reads.
O_NOINHERIT	indicates that this file is not to be inherited by a child process.
O_EXCL	indicates that this file is to be opened for exclusive access. If the file exists and `O_CREAT` was also specified then the open will fail (i.e., use `O_EXCL` to ensure that the file does not already exist).

When neither `O_TEXT` nor `O_BINARY` are specified, the default value in the global variable `_fmode` is used to set the file translation mode. When the program begins execution, this variable has a value of `O_TEXT.`

`O_CREAT` must be specified when the file does not exist and it is to be written.

When the file is to be created (`O_CREAT` is specified), an additional argument must be passed which contains the file permissions to be used for the new file. The access permissions for the file or directory are specified as a combination of bits (defined in the `<sys\stat.h>` header file).

The following bits define permissions for the owner.

Permission	***Meaning***
S_IRWXU	Read, write, execute/search
S_IRUSR	Read permission
S_IWUSR	Write permission
S_IXUSR	Execute/search permission

The following bits define permissions for the group.

Permission	***Meaning***
S_IRWXG	Read, write, execute/search
S_IRGRP	Read permission
S_IWGRP	Write permission
S_IXGRP	Execute/search permission

The following bits define permissions for others.

Permission	***Meaning***
S_IRWXO	Read, write, execute/search
S_IROTH	Read permission
S_IWOTH	Write permission
S_IXOTH	Execute/search permission

The following bits define miscellaneous permissions used by other implementations.

Permission	***Meaning***
S_IREAD	is equivalent to S_IRUSR (read permission)
S_IWRITE	is equivalent to S_IWUSR (write permission)
S_IEXEC	is equivalent to S_IXUSR (execute/search permission)

All files are readable with DOS; however, it is a good idea to set `S_IREAD` when read permission is intended for the file.

The `open` function applies the current file permission mask to the specified permissions (see `umask`).

Returns: If successful, `open` returns a handle for the file. When an error occurs while opening the file, -1 is returned.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EACCES	Access denied because *path* specifies a directory or a volume ID, or attempting to open a read-only file for writing
EMFILE	No more handles available (too many open files)
ENOENT	Path or file not found

See Also: `chsize`, `close`, `creat`, `dup`, `dup2`, `eof`, `exec` Functions, `fdopen`, `filelength`, `fileno`, `fstat`, `_grow_handles`, `isatty`, `lseek`, `read`, `setmode`, `sopen`, `stat`, `tell`, `write`, `umask`

Example:

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

void main()
  {
    int handle;

    /* open a file for output                  */
    /* replace existing file if it exists      */

    handle = open( "file",
                O_WRONLY | O_CREAT | O_TRUNC,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    /* read a file which is assumed to exist   */

    handle = open( "file", O_RDONLY );

    /* append to the end of an existing file   */
    /* write a new file if file does not exist */

    handle = open( "file",
                O_WRONLY | O_CREAT | O_APPEND,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
  }
```

Classification: open is POSIX 1003.1, _open is not POSIX, _wopen is not POSIX

_open conforms to ANSI/ISO naming conventions

Systems:

```
open - All, Netware
_open - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wopen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <direct.h>
struct dirent *opendir( const char *dirname );
struct _wdirent *_wopendir( const wchar_t *dirname );
```

Description: The `opendir` function is used in conjunction with the functions `readdir` and `closedir` to obtain the list of file names contained in the directory specified by *dirname*. The path indicated by *dirname* can be either relative to the current working directory or it can be an absolute path name. As an extension to POSIX, the last part of *dirname* can contain the characters '?' and '*' for matching multiple files within a directory.

The file `<direct.h>` contains definitions for the structure `dirent.`

```
#if defined(__OS2__) || defined(__NT__)
#define NAME_MAX 255    /* maximum for HPFS or NTFS */
#else
#define NAME_MAX  12    /* 8 chars + '.' +  3 chars */
#endif

typedef struct dirent {
    char    d_dta[ 21 ];        /* disk transfer area */
    char    d_attr;             /* file's attribute */
    unsigned short int d_time;  /* file's time */
    unsigned short int d_date;  /* file's date */
    long    d_size;             /* file's size */
    char    d_name[ NAME_MAX + 1 ]; /* file's name */
    unsigned short d_ino;       /* serial number */
    char    d_first;            /* flag for 1st time */
} DIR;
```

The file attribute field `d_attr` field is a set of bits representing the following attributes.

```
_A_RDONLY       /* Read-only file */
_A_HIDDEN       /* Hidden file */
_A_SYSTEM       /* System file */
_A_VOLID        /* Volume-ID entry (only MSFT knows) */
_A_SUBDIR       /* Subdirectory */
_A_ARCH         /* Archive file */
```

If the `_A_RDONLY` bit is off, then the file is read/write.

The format of the `d_time` field is described by the following structure (this structure is not defined in any Watcom header file).

```
typedef struct {
    unsigned short  twosecs : 5;    /* seconds / 2 */
    unsigned short  minutes : 6;    /* minutes (0,59) */
    unsigned short  hours   : 5;    /* hours (0,23) */
} ftime_t;
```

The format of the `d_date` field is described by the following structure (this structure is not defined in any Watcom header file).

```
typedef struct {
    unsigned short  day     : 5;    /* day (1,31) */
    unsigned short  month   : 4;    /* month (1,12) */
    unsigned short  year    : 7;    /* 0 is 1980 */
} fdate_t;
```

See the sample program below for an example of the use of these structures.

More than one directory can be read at the same time using the `opendir`, `readdir`, and `closedir` functions.

The `_wopendir` function is identical to `opendir` except that it accepts a wide-character string argument and returns a pointer to a `_wdirent` structure that can be used with the `_wreaddir` and `_wclosedir` functions.

The file `<direct.h>` contains definitions for the structure `_wdirent.`

```
struct _wdirent {
    char    d_dta[21];      /* disk transfer area */
    char    d_attr;         /* file's attribute */
    unsigned short int d_time;/* file's time */
    unsigned short int d_date;/* file's date */
    long    d_size;         /* file's size */
    wchar_t d_name[NAME_MAX+1];/* file's name */
    unsigned short d_ino;   /* serial number (not used) */
    char    d_first;        /* flag for 1st time */
};
```

Returns: The `opendir` function, if successful, returns a pointer to a structure required for subsequent calls to `readdir` to retrieve the file names matching the pattern specified by *dirname.* The `opendir` function returns `NULL` if *dirname* is not a valid pathname, or if there are no files matching *dirname.*

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EACCES	Search permission is denied for a component of *dirname* or read permission is denied for *dirname*.
ENOENT	The named directory does not exist.

See Also: `closedir`, `_dos_find` Functions, `readdir`, `rewinddir`

Example: To get a list of files contained in the directory `\watcom\h` on your default disk:

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

typedef struct {
    unsigned short  twosecs : 5;    /* seconds / 2 */
    unsigned short  minutes : 6;
    unsigned short  hours   : 5;
} ftime_t;

typedef struct {
    unsigned short  day     : 5;
    unsigned short  month   : 4;
    unsigned short  year    : 7;
} fdate_t;

void main()
  {
    DIR *dirp;
    struct dirent *direntp;
    ftime_t *f_time;
    fdate_t *f_date;
    dirp = opendir( "\\watcom\\h" );
    if( dirp != NULL ) {
      for(;;) {
        direntp = readdir( dirp );
        if( direntp == NULL ) break;
        f_time = (ftime_t *)&direntp->d_time;
        f_date = (fdate_t *)&direntp->d_date;
        printf( "%-12s %d/%2.2d/%2.2d "
                "%2.2d:%2.2d:%2.2d \n",
            direntp->d_name,
            f_date->year + 1980,
            f_date->month,
            f_date->day,
            f_time->hours,
            f_time->minutes,
            f_time->twosecs * 2 );
      }
      closedir( dirp );
    }
  }
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: opendir is POSIX 1003.1, _wopendir is not POSIX

Systems:
```
opendir - All, Netware
_wopendir - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <io.h>
int _open_osfhandle( long osfhandle, int access );
```

Description: The `_open_osfhandle` function allocates a POSIX-level file handle and sets it to point to the operating system's internal file handle specified by *osfhandle*. The value returned by `_get_osfhandle` can be used as an argument to the `_open_osfhandle` function.

The access mode is established by a combination of the bits defined in the `<fcntl.h>` header file. The following bits may be set:

Mode	*Meaning*
O_RDONLY	permit the file to be only read.
O_WRONLY	permit the file to be only written.
O_RDWR	permit the file to be both read and written.
O_APPEND	causes each record that is written to be written at the end of the file.
O_CREAT	has no effect when the file indicated by *filename* already exists; otherwise, the file is created;
O_TRUNC	causes the file to be truncated to contain no data when the file exists; has no effect when the file does not exist.
O_BINARY	causes the file to be opened in binary mode which means that data will be transmitted to and from the file unchanged.
O_TEXT	causes the file to be opened in text mode which means that carriage-return characters are written before any linefeed character that is written and causes carriage-return characters to be removed when encountered during reads.
O_NOINHERIT	indicates that this file is not to be inherited by a child process.
O_EXCL	indicates that this file is to be opened for exclusive access. If the file exists and `O_CREAT` was also specified then the open will fail (i.e., use `O_EXCL` to ensure that the file does not already exist).

When neither O_TEXT nor O_BINARY are specified, the default value in the global variable _fmode is used to set the file translation mode. When the program begins execution, this variable has a value of O_TEXT.

O_CREAT must be specified when the file does not exist and it is to be written.

When two or more manifest constants are used to form the *flags* argument, the constants are combined with the bitwise-OR operator (|).

The example below demonstrates the use of the _get_osfhandle and _open_osfhandle functions. Note that the example shows how the dup2 function can be used to obtain almost identical functionality.

When the POSIX-level file handles associated with one OS file handle are closed, the first one closes successfully but the others return an error (since the first call close the file and released the OS file handle). So it is important to call close at the right time, i.e., after all I/O operations are completed to the file.

Returns: If successful, _open_osfhandle returns a POSIX-style file handle. Otherwise, it returns -1.

See Also: close, _dos_open, dup2, fdopen, fopen, freopen, _fsopen, _get_osfhandle, _grow_handles, _hdopen, open, _os_handle, _popen, sopen

Example:

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

void main()
{
    long os_handle;
    int fh1, fh2, rc;

    fh1 = open( "file",
                O_WRONLY | O_CREAT | O_TRUNC | O_BINARY,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    if( fh1 == -1 ) {
        printf( "Could not open output file\n" );
        exit( EXIT_FAILURE );
    }
    printf( "First POSIX handle %d\n", fh1 );
#if defined(USE_DUP2)
    fh2 = 6;
    if( dup2( fh1, fh2 ) == -1 ) fh2 = -1;
#else
    os_handle = _get_osfhandle( fh1 );
    printf( "OS Handle %ld\n", os_handle );

    fh2 = _open_osfhandle( os_handle, O_WRONLY |
                                      O_BINARY );
#endif
    if( fh2 == -1 ) {
        printf( "Could not open with second handle\n" );
        exit( EXIT_FAILURE );
    }
    printf( "Second POSIX handle %d\n", fh2 );

    rc = write( fh2, "trash\x0d\x0a", 7 );
    printf( "Write file using second handle %d\n", rc );

    rc = close( fh2 );
    printf( "Closing second handle %d\n", rc );
    rc = close( fh1 );
    printf( "Closing first handle %d\n", rc );
}
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <io.h>
int _os_handle( int handle );
```

Description: The `_os_handle` function takes a POSIX-style file handle specified by *handle*. It returns the corresponding operating system level handle.

Returns: The `_os_handle` function returns the operating system handle that corresponds to the specified POSIX-style file handle.

See Also: `close`, `fdopen`, `_get_osfhandle`, `_hdopen`, `open`, `_open_osfhandle`

Example:
```
#include <stdio.h>
#include <io.h>

void main()
  {
    int handle;
    FILE *fp;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      handle = _os_handle( fileno( fp ) );
      fclose( fp );
    }
  }
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32, Netware

Synopsis:

```
#include <graph.h>
void _FAR _outgtext( char _FAR *text );
```

Description: The `_outgtext` function displays the character string indicated by the argument *text*. The string must be terminated by a null character ('\0').

The string is displayed starting at the current position (see the `_moveto` function) in the current color and in the currently selected font (see the `_setfont` function). The current position is updated to follow the displayed text.

When no font has been previously selected with `_setfont,` a default font will be used. The default font is an 8-by-8 bit-mapped font.

The graphics library can display text in three different ways.

1. The `_outtext` and `_outmem` functions can be used in any video mode. However, this variety of text can be displayed in only one size.

2. The `_grtext` function displays text as a sequence of line segments, and can be drawn in different sizes, with different orientations and alignments.

3. The `_outgtext` function displays text in the currently selected font. Both bit-mapped and vector fonts are supported; the size and type of text depends on the fonts that are available.

Returns: The `_outgtext` function does not return a value.

See Also: `_registerfonts`, `_unregisterfonts`, `_setfont`, `_getfontinfo`, `_getgtextextent`, `_setgtextvector`, `_getgtextvector`, `_outtext`, `_outmem`, `_grtext`

Example:

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

main()
{
    int i, n;
    char buf[ 10 ];

    _setvideomode( _VRES16COLOR );
    n = _registerfonts( "*.fon" );
    for( i = 0; i < n; ++i ) {
        sprintf( buf, "n%d", i );
        _setfont( buf );
        _moveto( 100, 100 );
        _outgtext( "WATCOM Graphics" );
        getch();
        _clearscreen( _GCLEARSCREEN );
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
void _FAR _outmem( char _FAR *text, short length );
```

Description: The `_outmem` function displays the character string indicated by the argument *text*. The argument *length* specifies the number of characters to be displayed. Unlike the `_outtext` function, `_outmem` will display the graphical representation of characters such as ASCII 10 and 0, instead of interpreting them as control characters.

The text is displayed using the current text color (see the `_settextcolor` function), starting at the current text position (see the `_settextposition` function). The text position is updated to follow the end of the displayed text.

The graphics library can display text in three different ways.

1. The `_outtext` and `_outmem` functions can be used in any video mode. However, this variety of text can be displayed in only one size.

2. The `_grtext` function displays text as a sequence of line segments, and can be drawn in different sizes, with different orientations and alignments.

3. The `_outgtext` function displays text in the currently selected font. Both bit-mapped and vector fonts are supported; the size and type of text depends on the fonts that are available.

Returns: The `_outmem` function does not return a value.

See Also: `_settextcolor`, `_settextposition`, `_settextwindow`, `_grtext`, `_outtext`, `_outgtext`

Example:

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

main()
{
    int i;
    char buf[ 1 ];

    _clearscreen( _GCLEARSCREEN );
    for( i = 0; i <= 255; ++i ) {
        _settextposition( 1 + i % 16,
                          1 + 5 * ( i / 16 ) );
        buf[ 0 ] = i;
        _outmem( buf, 1 );
    }
    getch();
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <conio.h>
unsigned int outp( int port, int value );
```

Description: The `outp` function writes one byte, determined by *value,* to the 80x86 hardware port whose number is given by *port.*

A hardware port is used to communicate with a device. One or two bytes can be read and/or written from each port, depending upon the hardware. Consult the technical documentation for your computer to determine the port numbers for a device and the expected usage of each port for a device.

Returns: The value transmitted is returned.

See Also: `inp`, `inpd`, `inpw`, `outpd`, `outpw`

Example:

```
#include <conio.h>

void main()
  {
    /* turn off speaker */
    outp( 0x61, inp( 0x61 ) & 0xFC );
  }
```

Classification: Intel

Systems: All, Netware

Synopsis:
```
#include <conio.h>
unsigned long outpd( int port,
                     unsigned long value );
```

Description: The `outpd` function writes a double-word (four bytes), determined by *value,* to the 80x86 hardware port whose number is given by *port.*

A hardware port is used to communicate with a device. One or two bytes can be read and/or written from each port, depending upon the hardware. Consult the technical documentation for your computer to determine the port numbers for a device and the expected usage of each port for a device.

Returns: The value transmitted is returned.

See Also: `inp`, `inpd`, `inpw`, `outp`, `outpw`

Example:
```
#include <conio.h>
#define DEVICE 34

void main()
  {
    outpd( DEVICE, 0x12345678 );
  }
```

Classification: Intel

Systems: DOS/32, Win386, Win32, QNX/32, OS/2-32, Netware

Synopsis:
```
#include <conio.h>
unsigned int outpw( int port,
                    unsigned int value );
```

Description: The `outpw` function writes a word (two bytes), determined by *value*, to the 80x86 hardware port whose number is given by *port*.

A hardware port is used to communicate with a device. One or two bytes can be read and/or written from each port, depending upon the hardware. Consult the technical documentation for your computer to determine the port numbers for a device and the expected usage of each port for a device.

Returns: The value transmitted is returned.

See Also: `inp`, `inpd`, `inpw`, `outp`, `outpd`

Example:
```
#include <conio.h>
#define DEVICE 34

void main()
  {
    outpw( DEVICE, 0x1234 );
  }
```

Classification: Intel

Systems: All, Netware

Synopsis:
```
#include <graph.h>
void _FAR _outtext( char _FAR *text );
```

Description: The `_outtext` function displays the character string indicated by the argument *text*. The string must be terminated by a null character ('\0'). When a line-feed character ('\n') is encountered in the string, the characters following will be displayed on the next row of the screen.

The text is displayed using the current text color (see the `_settextcolor` function), starting at the current text position (see the `_settextposition` function). The text position is updated to follow the end of the displayed text.

The graphics library can display text in three different ways.

1. The `_outtext` and `_outmem` functions can be used in any video mode. However, this variety of text can be displayed in only one size.

2. The `_grtext` function displays text as a sequence of line segments, and can be drawn in different sizes, with different orientations and alignments.

3. The `_outgtext` function displays text in the currently selected font. Both bit-mapped and vector fonts are supported; the size and type of text depends on the fonts that are available.

Returns: The `_outtext` function does not return a value.

See Also: `_settextcolor`, `_settextposition`, `_settextwindow`, `_grtext`, `_outmem`, `_outgtext`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    _setvideomode( _TEXTC80 );
    _settextposition( 10, 30 );
    _outtext( "WATCOM Graphics" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <stdio.h>
int _pclose( FILE *fp );
```

Description: The `_pclose` function closes the pipe associated with *fp* and waits for the subprocess created by `_popen` to terminate.

Returns: The `_pclose` function returns the termination status of the command language interpreter. If an error occured, `_pclose` returns (-1) with `errno` set appropriately.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EINTR	The `_pclose` function was interrupted by a signal while waiting for the child process to terminate.
ECHILD	The `_pclose` function was unable to obtain the termination status of the child process.

See Also: `perror`, `_pipe`, `_popen`

Example: See example provided with `_popen.`

Classification: WATCOM

Systems: Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <stdio.h>
void perror( const char *prefix );
void _wperror( const wchar_t *prefix );
```

Description: The `perror` function prints, on the file designated by `stderr`, the error message corresponding to the error number contained in `errno.` The `perror` function writes first the string pointed to by *prefix* to stderr. This is followed by a colon (":"), a space, the string returned by `strerror(errno)`, and a newline character.

The `_wperror` function is identical to `perror` except that it accepts a wide-character string argument and produces wide-character output.

Returns: The `perror` function returns no value. Because `perror` uses the `fprintf` function, `errno` can be set when an error is detected during the execution of that function.

See Also: `clearerr`, `feof`, `ferror`, `strerror`

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

void main()
  {
    FILE *fp;

    fp = fopen( "data.fil", "r" );
    if( fp == NULL ) {
        perror( "Unable to open file" );
    }
  }
```

Classification: perror is ANSI, _wperror is not ANSI

Systems:
```
perror - All, Netware
_wperror - All
```

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_analyzechart( chartenv _FAR *env,
                             char _FAR * _FAR *cat,
                             float _FAR *values, short n );

short _FAR _pg_analyzechartms( chartenv _FAR *env,
                               char _FAR * _FAR *cat,
                               float _FAR *values,
                               short nseries,
                               short n, short dim,
                               char _FAR * _FAR *labels );
```

Description: The `_pg_analyzechart` functions analyze either a single-series or a multi-series bar, column or line chart. These functions calculate default values for chart elements without actually displaying the chart.

The `_pg_analyzechart` function analyzes a single-series bar, column or line chart. The chart environment structure *env* is filled with default values based on the type of chart and the values of the *cat* and *values* arguments. The arguments are the same as for the `_pg_chart` function.

The `_pg_analyzechartms` function analyzes a multi-series bar, column or line chart. The chart environment structure *env* is filled with default values based on the type of chart and the values of the *cat, values* and *labels* arguments. The arguments are the same as for the `_pg_chartms` function.

Returns: The `_pg_analyzechart` functions return zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_analyzepie`, `_pg_analyzescatter`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_analyzechart( &env,
                      categories, values, NUM_VALUES );
    /* use manual scaling */
    env.yaxis.autoscale = 0;
    env.yaxis.scalemin = 0.0;
    env.yaxis.scalemax = 100.0;
    env.yaxis.ticinterval = 25.0;
    _pg_chart( &env, categories, values, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: _pg_analyzechart is PC Graphics

Systems:

```
_pg_analyzechart - DOS, QNX
_pg_analyzechartms - DOS, QNX
```

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_analyzepie( chartenv _FAR *env,
                           char _FAR * _FAR *cat,
                           float _FAR *values,
                           short _FAR *explode, short n );
```

Description: The `_pg_analyzepie` function analyzes a pie chart. This function calculates default values for chart elements without actually displaying the chart.

The chart environment structure *env* is filled with default values based on the values of the *cat*, *values* and *explode* arguments. The arguments are the same as for the `_pg_chartpie` function.

Returns: The `_pg_analyzepie` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_analyzechart`, `_pg_analyzescatter`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

short explode[ NUM_VALUES ] = {
    1, 0, 0, 0
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_PIECHART, _PG_NOPERCENT );
    strcpy( env.maintitle.title, "Pie Chart" );
    env.legend.place = _PG_BOTTOM;
    _pg_analyzepie( &env, categories,
                    values, explode, NUM_VALUES );
    /* make legend window same width as data window */
    env.legend.autosize = 0;
    env.legend.legendwindow.x1 = env.datawindow.x1;
    env.legend.legendwindow.x2 = env.datawindow.x2;
    _pg_chartpie( &env, categories,
                  values, explode, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_analyzescatter( chartenv _FAR *env,
                               float _FAR *x,
                               float _FAR *y, short n );

short _FAR _pg_analyzescatterms(
                      chartenv _FAR *env,
                      float _FAR *x, float _FAR *y,
                      short nseries, short n, short dim,
                      char _FAR * _FAR *labels );
```

Description: The `_pg_analyzescatter` functions analyze either a single-series or a multi-series scatter chart. These functions calculate default values for chart elements without actually displaying the chart.

The `_pg_analyzescatter` function analyzes a single-series scatter chart. The chart environment structure *env* is filled with default values based on the values of the *x* and *y* arguments. The arguments are the same as for the `_pg_chartscatter` function.

The `_pg_analyzescatterms` function analyzes a multi-series scatter chart. The chart environment structure *env* is filled with default values based on the values of the *x*, *y* and *labels* arguments. The arguments are the same as for the `_pg_chartscatterms` function.

Returns: The `_pg_analyzescatter` functions return zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_analyzechart`, `_pg_analyzepie`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4
#define NUM_SERIES 2

char _FAR *labels[ NUM_SERIES ] = {
    "Jan", "Feb"
};

float x[ NUM_SERIES ][ NUM_VALUES ] = {
    5, 15, 30, 40, 10, 20, 30, 45
};

float y[ NUM_SERIES ][ NUM_VALUES ] = {
    10, 15, 30, 45, 40, 30, 15, 5
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_SCATTERCHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Scatter Chart" );
    _pg_analyzescatterms( &env, x, y, NUM_SERIES,
                          NUM_VALUES, NUM_VALUES, labels );
    /* display x-axis labels with 2 decimal places */
    env.xaxis.autoscale = 0;
    env.xaxis.ticdecimals = 2;
    _pg_chartscatterms( &env, x, y, NUM_SERIES,
                        NUM_VALUES, NUM_VALUES, labels );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems:

```
_pg_analyzescatter - DOS, QNX
_pg_analyzescatterms - DOS, QNX
```

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_chart( chartenv _FAR *env,
                      char _FAR * _FAR *cat,
                      float _FAR *values, short n );

short _FAR _pg_chartms( chartenv _FAR *env,
                        char _FAR * _FAR *cat,
                        float _FAR *values, short nseries,
                        short n, short dim,
                        char _FAR * _FAR *labels );
```

Description: The `_pg_chart` functions display either a single-series or a multi-series bar, column or line chart. The type of chart displayed and other chart options are contained in the *env* argument. The argument *cat* is an array of strings. These strings describe the categories against which the data in the *values* array is charted.

The `_pg_chart` function displays a bar, column or line chart from the single series of data contained in the *values* array. The argument *n* specifies the number of values to chart.

The `_pg_chartms` function displays a multi-series bar, column or line chart. The argument *nseries* specifies the number of series of data to chart. The argument *values* is assumed to be a two-dimensional array defined as follows:

```
float values[ nseries ][ dim ];
```

The number of values used from each series is given by the argument *n*, where *n* is less than or equal to *dim*. The argument *labels* is an array of strings. These strings describe each of the series and are used in the chart legend.

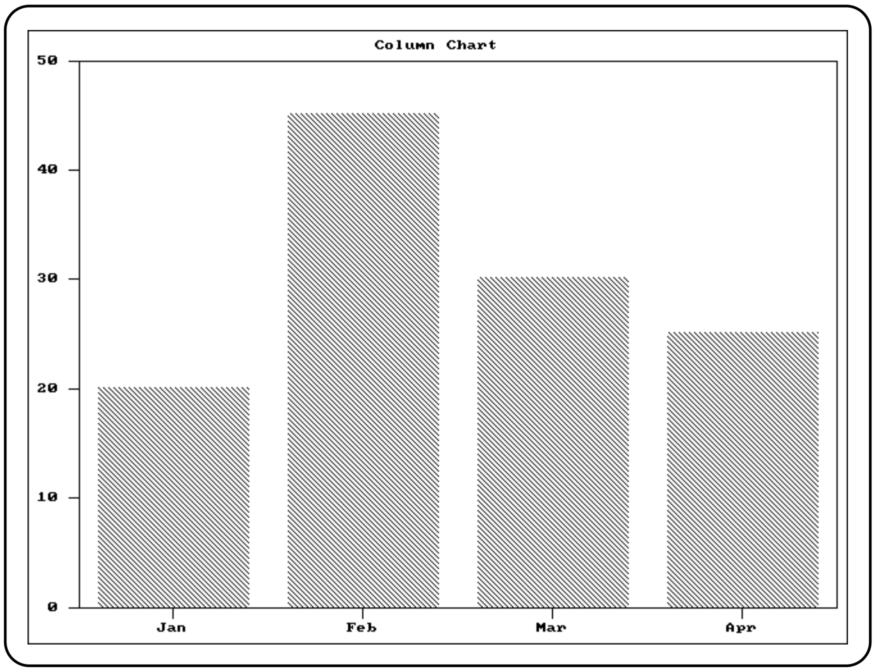
Returns: The `_pg_chart` functions return zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_analyzechart`, `_pg_analyzepie`, `_pg_analyzescatter`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_chart( &env, categories, values, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems:
```
_pg_chart - DOS, QNX
_pg_chartms - DOS, QNX
```

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_chartpie( chartenv _FAR *env,
                         char _FAR * _FAR *cat,
                         float _FAR *values,
                         short _FAR *explode, short n );
```

Description: The `_pg_chartpie` function displays a pie chart. The chart is displayed using the options specified in the *env* argument.

The pie chart is created from the data contained in the *values* array. The argument *n* specifies the number of values to chart.

The argument *cat* is an array of strings. These strings describe each of the pie slices and are used in the chart legend. The argument *explode* is an array of values corresponding to each of the pie slices. For each non-zero element in the array, the corresponding pie slice is drawn "exploded", or slightly offset from the rest of the pie.

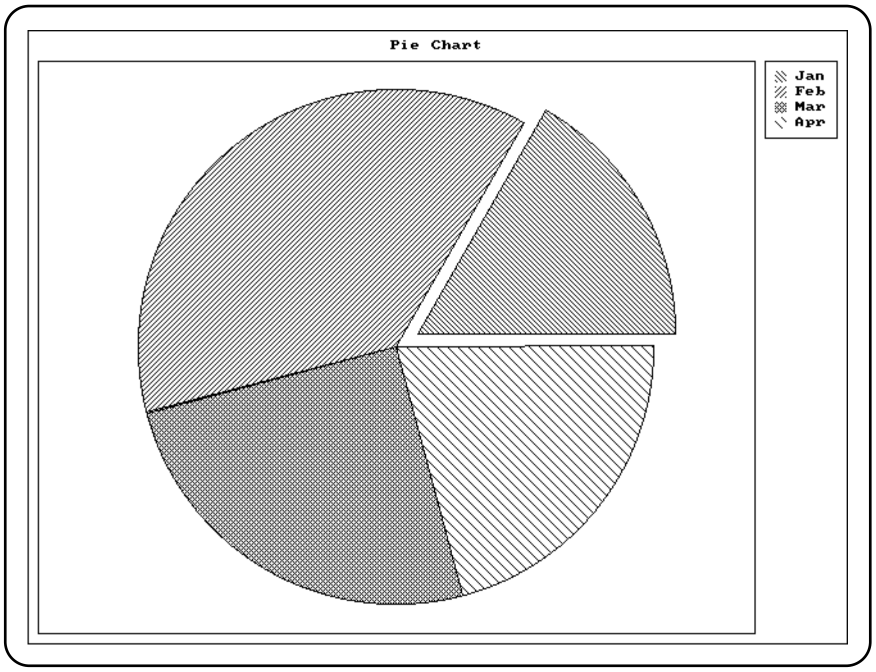
Returns: The `_pg_chartpie` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartscatter`, `_pg_analyzechart`, `_pg_analyzepie`, `_pg_analyzescatter`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

short explode[ NUM_VALUES ] = {
    1, 0, 0, 0
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_PIECHART, _PG_NOPERCENT );
    strcpy( env.maintitle.title, "Pie Chart" );
    _pg_chartpie( &env, categories,
                  values, explode, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_chartscatter( chartenv _FAR *env,
                             float _FAR *x,
                             float _FAR *y, short n );

short _FAR _pg_chartscatterms( chartenv _FAR *env,
                               float _FAR *x,
                               float _FAR *y,
                               short nseries,
                               short n, short dim,
                               char _FAR * _FAR *labels );
```

Description: The `_pg_chartscatter` functions display either a single-series or a multi-series scatter chart. The chart is displayed using the options specified in the *env* argument.

The `_pg_chartscatter` function displays a scatter chart from the single series of data contained in the arrays *x* and *y*. The argument *n* specifies the number of values to chart.

The `_pg_chartscatterms` function displays a multi-series scatter chart. The argument *nseries* specifies the number of series of data to chart. The arguments *x* and *y* are assumed to be two-dimensional arrays defined as follows:

```
float x[ nseries ][ dim ];
```

The number of values used from each series is given by the argument *n*, where *n* is less than or equal to *dim*. The argument *labels* is an array of strings. These strings describe each of the series and are used in the chart legend.

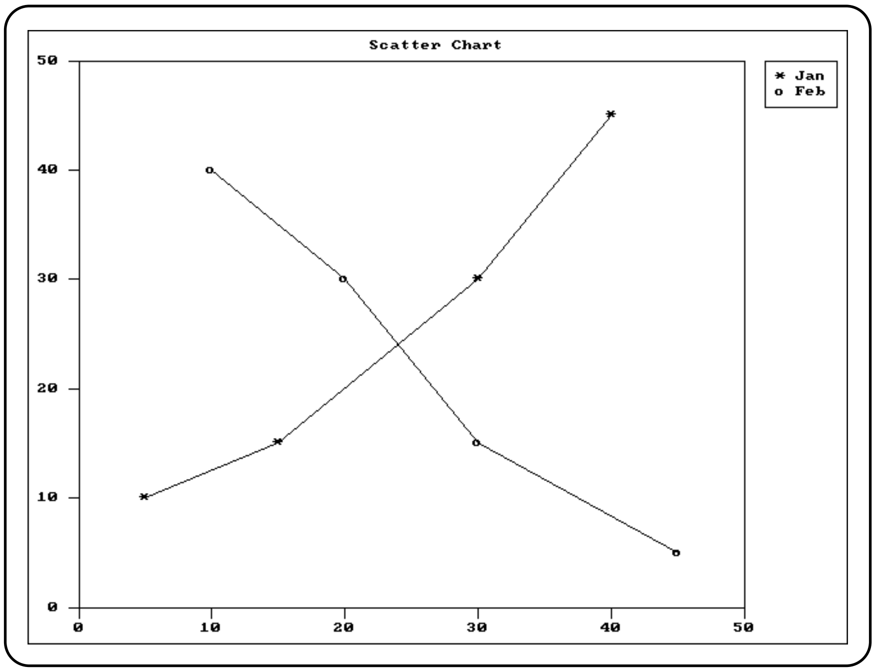
Returns: The `_pg_chartscatter` functions return zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_analyzechart`, `_pg_analyzepie`, `_pg_analyzescatter`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4
#define NUM_SERIES 2

char _FAR *labels[ NUM_SERIES ] = {
    "Jan", "Feb"
};

float x[ NUM_SERIES ][ NUM_VALUES ] = {
    5, 15, 30, 40, 10, 20, 30, 45
};

float y[ NUM_SERIES ][ NUM_VALUES ] = {
    10, 15, 30, 45, 40, 30, 15, 5
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_SCATTERCHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Scatter Chart" );
    _pg_chartscatterms( &env, x, y, NUM_SERIES,
                        NUM_VALUES, NUM_VALUES, labels );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: _pg_chartscatter - DOS, QNX
_pg_chartscatterms - DOS, QNX

Synopsis:
```
#include <pgchart.h>
short _FAR _pg_defaultchart( chartenv _FAR *env,
                             short type, short style );
```

Description: The `_pg_defaultchart` function initializes the chart structure *env* to contain default values before a chart is drawn. All values in the chart structure are initialized, including blanking of all titles. The chart type in the structure is initialized to the value *type,* and the chart style is initialized to *style.*

The argument *type* can have one of the following values:

_PG_BARCHART	Bar chart (horizontal bars)
_PG_COLUMNCHART	Column chart (vertical bars)
_PG_LINECHART	Line chart
_PG_SCATTERCHART	Scatter chart
_PG_PIECHART	Pie chart

Each type of chart can be drawn in one of two styles. For each chart type the argument *style* can have one of the following values:

```
Type            Style 1                 Style 2

Bar             _PG_PLAINBARS           _PG_STACKEDBARS
Column          _PG_PLAINBARS           _PG_STACKEDBARS
Line            _PG_POINTANDLINE        _PG_POINTONLY
Scatter         _PG_POINTANDLINE        _PG_POINTONLY
Pie             _PG_PERCENT             _PG_NOPERCENT
```

For single-series bar and column charts, the chart style is ignored. The "plain" (clustered) and "stacked" styles only apply when there is more than one series of data. The "percent" style for pie charts causes percentages to be displayed beside each of the pie slices.

Returns: The `_pg_defaultchart` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`

Example:
```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_chart( &env, categories, values, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_getchardef( short ch,
                           unsigned char _FAR *def );
```

Description: The `_pg_getchardef` function retrieves the current bit-map definition for the character *ch*. The bit-map is placed in the array *def*. The current font must be an 8-by-8 bit-mapped font.

Returns: The `_pg_getchardef` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_setchardef`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#define NUM_VALUES 4

float x[ NUM_VALUES ] = {
    5, 25, 45, 65
};

float y[ NUM_VALUES ] = {
    5, 45, 25, 65
};

char diamond[ 8 ] = {
    0x10, 0x28, 0x44, 0x82, 0x44, 0x28, 0x10, 0x00
};

main()
{
    chartenv env;
    char old_def[ 8 ];

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_SCATTERCHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Scatter Chart" );
    /* change asterisk character to diamond */
    _pg_getchardef( '*', old_def );
    _pg_setchardef( '*', diamond );
    _pg_chartscatter( &env, x, y, NUM_VALUES );
    _pg_setchardef( '*', old_def );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_getpalette( paletteentry _FAR *pal );
```

Description: The `_pg_getpalette` function retrieves the internal palette of the presentation graphics system. The palette controls the colors, line styles, fill patterns and plot characters used to display each series of data in a chart.

The argument *pal* is an array of palette structures that will contain the palette. Each element of the palette is a structure containing the following fields:

color	color used to display series
style	line style used for line and scatter charts
fill	fill pattern used to fill interior of bar and pie sections
plotchar	character plotted on line and scatter charts

Returns: The `_pg_getpalette` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_setpalette`, `_pg_resetpalette`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

char bricks[ 8 ] = {
    0xff, 0x80, 0x80, 0x80, 0xff, 0x08, 0x08, 0x08
};

main()
{
    chartenv env;
    palettetype pal;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* get default palette and change 1st entry */
    _pg_getpalette( &pal );
    pal[ 1 ].color = 12;
    memcpy( pal[ 1 ].fill, bricks, 8 );
    /* use new palette */
    _pg_setpalette( &pal );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset palette to default */
    _pg_resetpalette();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
void _FAR _pg_getstyleset( unsigned short _FAR *style );
```

Description: The `_pg_getstyleset` function retrieves the internal style-set of the presentation graphics system. The style-set is a set of line styles used for drawing window borders and grid-lines. The argument *style* is an array that will contain the style-set.

Returns: The `_pg_getstyleset` function does not return a value.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_setstyleset`, `_pg_resetstyleset`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;
    styleset style;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* turn on yaxis grid, and use style 2 */
    env.yaxis.grid = 1;
    env.yaxis.gridstyle = 2;
    /* get default style-set and change entry 2 */
    _pg_getstyleset( &style );
    style[ 2 ] = 0x8888;
    /* use new style-set */
    _pg_setstyleset( &style );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset style-set to default */
    _pg_resetstyleset();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_hlabelchart( chartenv _FAR *env,
                            short x, short y,
                            short color,
                            char _FAR *label );
```

Description: The `_pg_hlabelchart` function displays the text string *label* on the chart described by the *env* chart structure. The string is displayed horizontally starting at the point `(x,y)`, relative to the upper left corner of the chart. The *color* specifies the palette color used to display the string.

Returns: The `_pg_hlabelchart` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_vlabelchart`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_chart( &env, categories, values, NUM_VALUES );
    _pg_hlabelchart( &env, 64, 32, 1, "Horizontal label" );
    _pg_vlabelchart( &env, 48, 32, 1, "Vertical label" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_initchart( void );
```

Description: The `_pg_initchart` function initializes the presentation graphics system. This includes initializing the internal palette and style-set used when drawing charts. This function must be called before any of the other presentation graphics functions.

The initialization of the presentation graphics system requires that a valid graphics mode has been selected. For this reason the `_setvideomode` function must be called before `_pg_initchart` is called. If a font has been selected (with the `_setfont` function), that font will be used when text is displayed in a chart. Font selection should also be done before initializing the presentation graphics system.

Returns: The `_pg_initchart` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_setvideomode`, `_setfont`, `_registerfonts`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_chart( &env, categories, values, NUM_VALUES );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
short _FAR _pg_resetpalette( void );
```

Description: The `_pg_resetpalette` function resets the internal palette of the presentation graphics system to default values. The palette controls the colors, line styles, fill patterns and plot characters used to display each series of data in a chart. The default palette chosen is dependent on the current video mode.

Returns: The `_pg_resetpalette` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_getpalette`, `_pg_setpalette`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

char bricks[ 8 ] = {
    0xff, 0x80, 0x80, 0x80, 0xff, 0x08, 0x08, 0x08
};

main()
{
    chartenv env;
    palettetype pal;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* get default palette and change 1st entry */
    _pg_getpalette( &pal );
    pal[ 1 ].color = 12;
    memcpy( pal[ 1 ].fill, bricks, 8 );
    /* use new palette */
    _pg_setpalette( &pal );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset palette to default */
    _pg_resetpalette();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
void _FAR _pg_resetstyleset( void );
```

Description: The `_pg_resetstyleset` function resets the internal style-set of the presentation graphics system to default values. The style-set is a set of line styles used for drawing window borders and grid-lines.

Returns: The `_pg_resetstyleset` function does not return a value.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_getstyleset`, `_pg_setstyleset`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;
    styleset style;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* turn on yaxis grid, and use style 2 */
    env.yaxis.grid = 1;
    env.yaxis.gridstyle = 2;
    /* get default style-set and change entry 2 */
    _pg_getstyleset( &style );
    style[ 2 ] = 0x8888;
    /* use new style-set */
    _pg_setstyleset( &style );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset style-set to default */
    _pg_resetstyleset();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
short _FAR _pg_setchardef( short ch,
                           unsigned char _FAR *def );
```

Description: The `_pg_setchardef` function sets the current bit-map definition for the character *ch*. The bit-map is contained in the array *def*. The current font must be an 8-by-8 bit-mapped font.

Returns: The `_pg_setchardef` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_getchardef`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#define NUM_VALUES 4

float x[ NUM_VALUES ] = {
    5, 25, 45, 65
};

float y[ NUM_VALUES ] = {
    5, 45, 25, 65
};

char diamond[ 8 ] = {
    0x10, 0x28, 0x44, 0x82, 0x44, 0x28, 0x10, 0x00
};

main()
{
    chartenv env;
    char old_def[ 8 ];

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_SCATTERCHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Scatter Chart" );
    /* change asterisk character to diamond */
    _pg_getchardef( '*', old_def );
    _pg_setchardef( '*', diamond );
    _pg_chartscatter( &env, x, y, NUM_VALUES );
    _pg_setchardef( '*', old_def );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
short _FAR _pg_setpalette( paletteentry _FAR *pal );
```

Description: The `_pg_setpalette` function sets the internal palette of the presentation graphics system. The palette controls the colors, line styles, fill patterns and plot characters used to display each series of data in a chart.

The argument *pal* is an array of palette structures containing the new palette. Each element of the palette is a structure containing the following fields:

color	color used to display series
style	line style used for line and scatter charts
fill	fill pattern used to fill interior of bar and pie sections
plotchar	character plotted on line and scatter charts

Returns: The `_pg_setpalette` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_getpalette`, `_pg_resetpalette`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

char bricks[ 8 ] = {
    0xff, 0x80, 0x80, 0x80, 0xff, 0x08, 0x08, 0x08
};

main()
{
    chartenv env;
    palettetype pal;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* get default palette and change 1st entry */
    _pg_getpalette( &pal );
    pal[ 1 ].color = 12;
    memcpy( pal[ 1 ].fill, bricks, 8 );
    /* use new palette */
    _pg_setpalette( &pal );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset palette to default */
    _pg_resetpalette();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <pgchart.h>
void _FAR _pg_setstyleset( unsigned short _FAR *style );
```

Description: The `_pg_setstyleset` function retrieves the internal style-set of the presentation graphics system. The style-set is a set of line styles used for drawing window borders and grid-lines. The argument *style* is an array containing the new style-set.

Returns: The `_pg_setstyleset` function does not return a value.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_getstyleset`, `_pg_resetstyleset`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;
    styleset style;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    /* turn on yaxis grid, and use style 2 */
    env.yaxis.grid = 1;
    env.yaxis.gridstyle = 2;
    /* get default style-set and change entry 2 */
    _pg_getstyleset( &style );
    style[ 2 ] = 0x8888;
    /* use new style-set */
    _pg_setstyleset( &style );
    _pg_chart( &env, categories, values, NUM_VALUES );
    /* reset style-set to default */
    _pg_resetstyleset();
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <pgchart.h>
short _FAR _pg_vlabelchart( chartenv _FAR *env,
                            short x, short y,
                            short color,
                            char _FAR *label );
```

Description: The `_pg_vlabelchart` function displays the text string *label* on the chart described by the *env* chart structure. The string is displayed vertically starting at the point `(x,y)`, relative to the upper left corner of the chart. The *color* specifies the palette color used to display the string.

Returns: The `_pg_vlabelchart` function returns zero if successful; otherwise, a non-zero value is returned.

See Also: `_pg_defaultchart`, `_pg_initchart`, `_pg_chart`, `_pg_chartpie`, `_pg_chartscatter`, `_pg_hlabelchart`

Example:

```
#include <graph.h>
#include <pgchart.h>
#include <string.h>
#include <conio.h>

#if defined ( __386__ )
    #define _FAR
#else
    #define _FAR    __far
#endif

#define NUM_VALUES 4

char _FAR *categories[ NUM_VALUES ] = {
    "Jan", "Feb", "Mar", "Apr"
};

float values[ NUM_VALUES ] = {
    20, 45, 30, 25
};

main()
{
    chartenv env;

    _setvideomode( _VRES16COLOR );
    _pg_initchart();
    _pg_defaultchart( &env,
                      _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Column Chart" );
    _pg_chart( &env, categories, values, NUM_VALUES );
    _pg_hlabelchart( &env, 64, 32, 1, "Horizontal label" );
    _pg_vlabelchart( &env, 48, 32, 1, "Vertical label" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

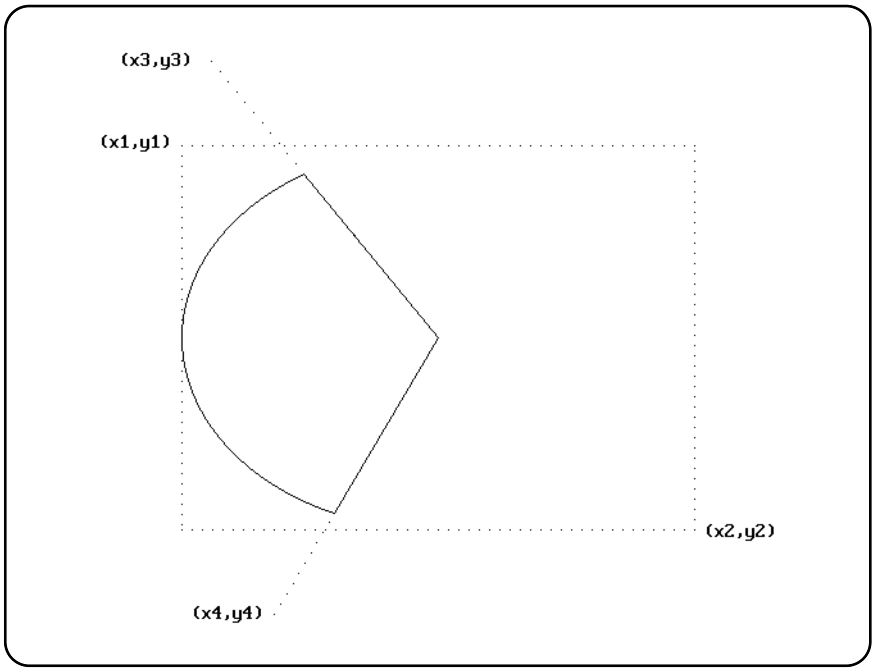
```
#include <graph.h>
short _FAR _pie( short fill, short x1, short y1,
                             short x2, short y2,
                             short x3, short y3,
                             short x4, short y4 );

short _FAR _pie_w( short fill, double x1, double y1,
                               double x2, double y2,
                               double x3, double y3,
                               double x4, double y4 );

short _FAR _pie_wxy( short fill,
                     struct _wxycoord _FAR *p1,
                     struct _wxycoord _FAR *p2,
                     struct _wxycoord _FAR *p3,
                     struct _wxycoord _FAR *p4 );
```

Description: The `_pie` functions draw pie-shaped wedges. The `_pie` function uses the view coordinate system. The `_pie_w` and `_pie_wxy` functions use the window coordinate system.

The pie wedges are drawn by drawing an elliptical arc (in the way described for the `_arc` functions) and then joining the center of the rectangle that contains the ellipse to the two endpoints of the arc.

The elliptical arc is drawn with its center at the center of the rectangle established by the points `(x1,y1)` and `(x2,y2)`. The arc is a segment of the ellipse drawn within this bounding rectangle. The arc starts at the point on this ellipse that intersects the vector from the centre of the ellipse to the point `(x3,y3)`. The arc ends at the point on this ellipse that intersects the vector from the centre of the ellipse to the point `(x4,y4)`. The arc is drawn in a counter-clockwise direction with the current plot action using the current color and the current line style.

The following picture illustrates the way in which the bounding rectangle and the vectors specifying the start and end points are defined.



When the coordinates (x1,y1) and (x2,y2) establish a line or a point (this happens when one or more of the x-coordinates or y-coordinates are equal), nothing is drawn.

The argument *fill* determines whether the figure is filled in or has only its outline drawn. The argument can have one of two values:

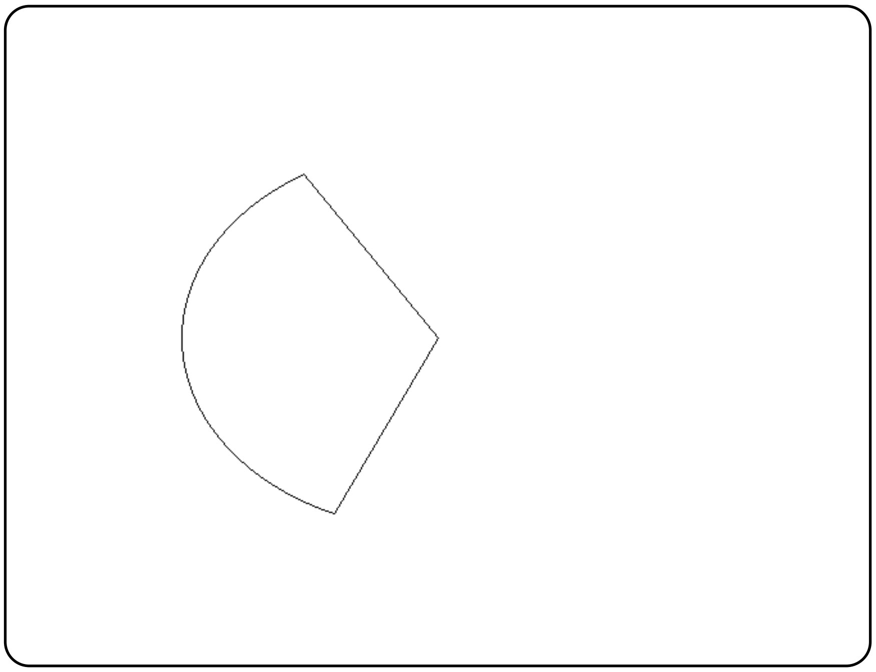
_GFILLINTERIOR	fill the interior by writing pixels with the current plot action using the current color and the current fill mask
_GBORDER	leave the interior unchanged; draw the outline of the figure with the current plot action using the current color and line style

Returns: The `_pie` functions return a non-zero value when the figure was successfully drawn; otherwise, zero is returned.

See Also: `_arc`, `_ellipse`, `_setcolor`, `_setfillmask`, `_setlinestyle`, `_setplotaction`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    _setvideomode( _VRES16COLOR );
    _pie( _GBORDER, 120, 90, 520, 390,
                    140, 20, 190, 460 );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems:
```
_pie - DOS, QNX
_pie_w - DOS, QNX
_pie_wxy - DOS, QNX
```

Synopsis:

```
#include <io.h>
int _pipe( int *phandles, unsigned psize, int textmode );
```

Description: The `_pipe` function creates a pipe (an unnamed FIFO) and places a file descriptor for the read end of the pipe in *phandles[0]* and a file descriptor for the write end of the pipe in *phandles[1]*. Their integer values are the two lowest available at the time of the `_pipe` function call. The `O_NONBLOCK` flag is cleared for both file descriptors. (The `fcntl` call can be used to set the `O_NONBLOCK` flag.)

Data can be written to file descriptor *phandles[1]* and read from file descriptor *phandles[0]*. A read on file descriptor *phandles[0]* returns the data written to *phandles[1]* on a first-in-first-out (FIFO) basis.

This function is typically used to connect together standard utilities to act as filters, passing the write end of the pipe to the data producing process as its `STDOUT_FILENO` and the read end of the pipe to the data consuming process as its `STDIN_FILENO.` (either via the traditional fork/dup2/exec or the more efficient spawn calls).

If successful, `_pipe` marks for update the *st_ftime*, *st_ctime*, *st_atime* and *st_mtime* fields of the pipe for updating.

Returns: The `_pipe` function returns zero on success. Otherwise, (-1) is returned and `errno` is set to indicate the error.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected. If any of the following conditions occur, the `_pipe` function shall return (-1) and set `errno` to the corresponding value:

Constant	***Meaning***
EMFILE	The calling process does not have at least 2 unused file descriptors available.
ENFILE	The number of simultaneously open files in the system would exceed the configured limit.
ENOSPC	There is insufficient space available to allocate the pipe buffer.
EROFS	The pipe pathname space is a read-only filesystem.

See Also: `open`, `_pclose`, `perror`, `_popen`, `read`, `write`

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <fcntl.h>
#include <io.h>
#include <dos.h>
#include <process.h>

static int handles[2] = { 0, 0 };
static int pid;

create_pipe()
  {
    if( _pipe( (int *)&handles, 2048, _O_BINARY ) == -1 ) {
      perror( "create_pipe" );
      exit( EXIT_FAILURE );
    }
  }

create_child( char *name )
  {
    char buff[10];

    itoa( handles[0], buff, 10 );
    pid = spawnl( P_NOWAIT, name,
                  "_pipe", buff, NULL );
    close( handles[0] );
    if( pid == -1 ) {
      perror( "create_child" );
      close( handles[1] );
      exit( EXIT_FAILURE );
    }
  }
```

```
fill_pipe()
  {
    int i;
    int rc;

    for( i = 1; i <= 10; i++ ) {
      printf( "Child, what is 5 times %d\n", i );
      rc = write( handles[1], &i, sizeof( int ) );
      if( rc < sizeof( int ) ) {
        perror( "fill_pipe" );
        close( handles[1] );
        exit( EXIT_FAILURE );
      }
    }
    /* indicate that we are done */
    write( handles[1], &i, 1 );
    close( handles[1] );
  }

empty_pipe( int in_pipe )
  {
    int i;
    int amt;

    for(;;) {
      amt = read( in_pipe, &i, sizeof( int ) );
      if( amt != sizeof( int ) ) break;
      printf( "Parent, 5 times %d is %d\n", i, 5*i );
    }
    if( amt == -1 ) {
      perror( "empty_pipe" );
      exit( EXIT_FAILURE );
    }
    close( in_pipe );
  }
```

```
void main( int argc, char *argv[] )
  {
    if( argc <= 1 ) {
      /* we are the spawning process */
      create_pipe();
      create_child( argv[0] );
      fill_pipe();
    } else {
      /* we are the spawned process */
      empty_pipe( atoi( argv[1] ) );
    }
    exit( EXIT_SUCCESS );
  }
```

produces the following:

```
Child, what is 5 times 1
Child, what is 5 times 2
Parent, 5 times 1 is 5
Parent, 5 times 2 is 10
Child, what is 5 times 3
Child, what is 5 times 4
Parent, 5 times 3 is 15
Parent, 5 times 4 is 20
Child, what is 5 times 5
Child, what is 5 times 6
Parent, 5 times 5 is 25
Parent, 5 times 6 is 30
Child, what is 5 times 7
Child, what is 5 times 8
Parent, 5 times 7 is 35
Parent, 5 times 8 is 40
Child, what is 5 times 9
Child, what is 5 times 10
Parent, 5 times 9 is 45
Parent, 5 times 10 is 50
```

Classification: WATCOM

Systems: Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <graph.h>
short _FAR _polygon( short fill, short numpts,
                     struct xycoord _FAR *points );

short _FAR _polygon_w( short fill, short numpts,
                       double _FAR *points );

short _FAR _polygon_wxy( short fill, short numpts,
                         struct _wxycoord _FAR *points );
```

Description: The `_polygon` functions draw polygons. The `_polygon` function uses the view coordinate system. The `_polygon_w` and `_polygon_wxy` functions use the window coordinate system.

The polygon is defined as containing *numpts* points whose coordinates are given in the array *points*.

The argument *fill* determines whether the polygon is filled in or has only its outline drawn. The argument can have one of two values:

_GFILLINTERIOR	fill the interior by writing pixels with the current plot action using the current color and the current fill mask
_GBORDER	leave the interior unchanged; draw the outline of the figure with the current plot action using the current color and line style

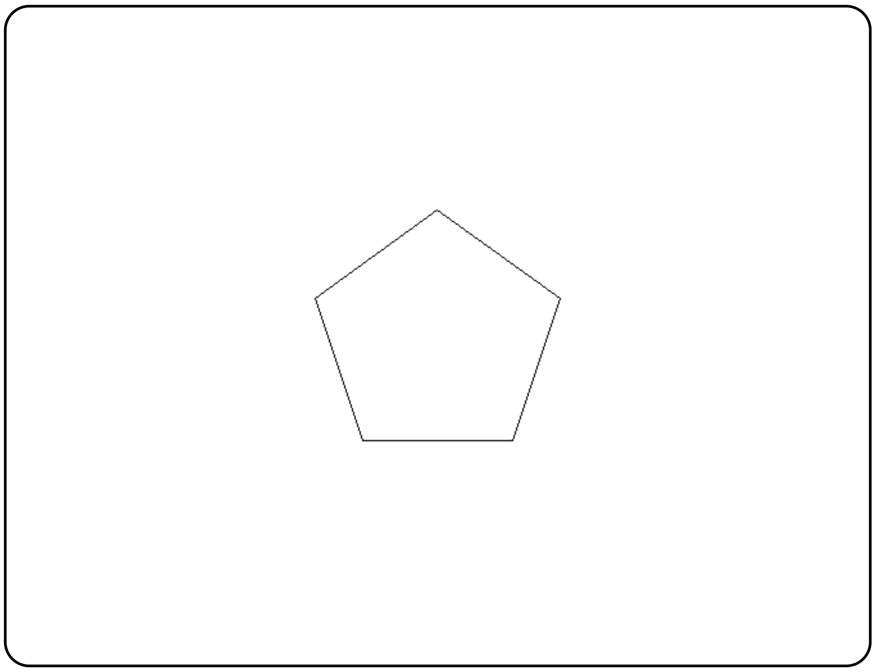
Returns: The `_polygon` functions return a non-zero value when the polygon was successfully drawn; otherwise, zero is returned.

See Also: `_setcolor`, `_setfillmask`, `_setlinestyle`, `_setplotaction`

Example:

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

struct xycoord points[ 5 ] = {
    319, 140, 224, 209, 261, 320,
    378, 320, 415, 209
};

main()
{
    _setvideomode( _VRES16COLOR );
    _polygon( _GBORDER, 5, points );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems:

```
_polygon - DOS, QNX
_polygon_w - DOS, QNX
```

```
_polygon_wxy - DOS, QNX
```

Synopsis:

```
#include <stdio.h>
FILE *_popen( const char *command, const char *mode );
FILE *_wpopen( const wchar_t *command, const wchar_t *mode );
```

Description: The `_popen` function executes the command specified by *command* and creates a pipe between the calling process and the executed command.

Depending on the *mode* argument, the stream pointer returned may be used to read from or write to the pipe.

The executed command has an environment the same as its parents. The command will be started as follows: spawnl(<shell_path>, <shell>, "-c", command, (char *)NULL);

where `<shell_path>` is an unspecified path for the shell utility and `<shell>` is one of "command.com" (DOS, Windows 95) or "cmd.exe" (Windows NT/2000, OS/2).

The *mode* argument to `_popen` is a string that specifies an I/O mode for the pipe.

Mode	***Meaning***
"r"	The calling process will read from the standard output of the child process using the stream pointer returned by `_popen`.
"w"	The calling process will write to the standard input of the child process using the stream pointer returned by `_popen`.

The letter "t" may be added to any of the above modes to indicate that the file is (or must be) a text file (i.e., CR/LF pairs are converted to newline characters).

The letter "b" may be added to any of the above modes to indicate that the file is (or must be) a binary file (an ANSI requirement for portability to systems that make a distinction between text and binary files).

When default file translation is specified (i.e., no "t" or "b" is specified), the value of the global variable `_fmode` establishes whether the file is to treated as a binary or a text file. Unless this value is changed by the program, the default will be text mode.

A stream opened by `_popen` should be closed by the `pclose` function.

Returns: The `_popen` function returns a non-NULL stream pointer upon successful completion. If `_popen` is unable to create either the pipe or the subprocess, a `NULL` stream pointer is returned and `errno` is set appropriately.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	*Meaning*
EINVAL	The *mode* argument is invalid.

`_popen` may also set `errno` values as described by the `_pipe` and `spawnl` functions.

See Also: `_grow_handles`, `_pclose`, `perror`, `_pipe`

Example:

```
/*
 * Executes a given program, converting all
 * output to upper case.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

char   buffer[256];

void main( int argc, char **argv )
  {
    int  i;
    int  c;
    FILE *f;

    for( i = 1; i < argc; i++ ) {
      strcat( buffer, argv[i] );
      strcat( buffer, " " );
    }

    if( ( f = _popen( buffer, "r" ) ) == NULL ) {
      perror( "_popen" );
      exit( 1 );
    }
    while( ( c = getc(f) ) != EOF ) {
      if( islower( c ) )
          c = toupper( c );
      putchar( c );
    }
    _pclose( f );
  }
```

Classification: WATCOM

Systems:
```
_popen - Win32, OS/2 1.x(all), OS/2-32
_wpopen - Win32, OS/2 1.x(all), OS/2-32
```

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

Description: The `pow` function computes *x* raised to the power *y*. A domain error occurs if *x* is zero and *y* is less than or equal to 0, or if *x* is negative and *y* is not an integer. A range error may occur.

Returns: The `pow` function returns the value of *x* raised to the power *y*. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `EDOM`, and print a "DOMAIN error" diagnostic message using the `stderr` stream.

See Also: `exp`, `log`, `sqrt`

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

void main()
  {
    printf( "%f\n", pow( 1.5, 2.5 ) );
  }
```

produces the following:

```
2.755676
```

Classification: ANSI

Systems: Math

Synopsis:

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

Description: The `printf` function writes output to the file designated by `stdout` under control of the argument *format.* The *format* string is described below.

The `wprintf` function is identical to `printf` except that it accepts a wide-character string argument for *format.*

Returns: The `printf` function returns the number of characters written, or a negative value if an output error occurred. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `sprintf`, `_vbprintf`, `vcprintf`, `vfprintf`, `vprintf`, `vsprintf`

Example:

```
#include <stdio.h>

void main()
  {
    char *weekday, *month;

    weekday = "Saturday";
    month = "April";
    printf( "%s, %s %d, %d\n",
          weekday, month, 18, 1987 );
    printf( "f1 = %8.4f f2 = %10.2E x = %#08x i = %d\n",
            23.45,      3141.5926,   0x1db,     -1 );
  }
```

produces the following:

```
Saturday, April 18, 1987
f1 =  23.4500 f2 =  3.14E+003 x = 0x0001db i = -1
```

Format Control String: The format control string consists of *ordinary characters,* that are written exactly as they occur in the format string, and *conversion specifiers,* that cause argument values to be written as they are encountered during the processing of the format string. An ordinary character in the format string is any character, other than a percent character (%), that is not part of a conversion specifier. A conversion specifier is a sequence of characters in the format string that begins with a percent character (%) and is followed, in sequence, by the following:

- zero or more *format control flags* that can modify the final effect of the format directive;
- an optional decimal integer, or an asterisk character ('*'), that specifies a *minimum field width* to be reserved for the formatted item;
- an optional *precision* specification in the form of a period character (.), followed by an optional decimal integer or an asterisk character (*);
- an optional *type length* specification: one of "h", "l", "L", "I64", "w", "N" or "F"; and
- a character that specifies the type of conversion to be performed: one of the characters "cCdeEfFgGinopsSuxX".

The valid format control flags are:

`"-"` the formatted item is left-justified within the field; normally, items are right-justified

`"+"` a signed, positive object will always start with a plus character (+); normally, only negative items begin with a sign

`" "` a signed, positive object will always start with a space character; if both "+" and " " are specified, "+" overrides " "

`"#"` an alternate conversion form is used:

- for "o" (unsigned octal) conversions, the precision is incremented, if necessary, so that the first digit is "0".
- for "x" or "X" (unsigned hexadecimal) conversions, a non-zero value is prepended with "0x" or "0X" respectively.
- for "e", "E", "f", "g" or "G" (any floating-point) conversions, the result always contains a decimal-point character, even if no digits follow it; normally, a decimal-point character appears in the result only if there is a digit to follow it.
- in addition to the preceding, for "g" or "G" conversions, trailing zeros are not removed from the result.

If no field width is specified, or if the value that is given is less than the number of characters in the converted value (subject to any precision value), a field of sufficient width to contain the converted value is used. If the converted value has fewer characters than are specified by the field width, the value is padded on the left (or right, subject to the left-justification flag) with spaces or zero characters ("0"). If the field width begins with "0" and no precision is specified, the value is padded with zeros; otherwise the value is padded with spaces. If the field width is "*", a value of type `int` from the argument list is used (before a precision argument or a conversion argument) as the minimum field width. A negative field width value is interpreted as a left-justification flag, followed by a positive field width.

As with the field width specifier, a precision specifier of "*" causes a value of type `int` from the argument list to be used as the precision specifier. If no precision value is given, a precision of 0 is used. The precision value affects the following conversions:

- For "d", "i", "o", "u", "x" and "X" (integer) conversions, the precision specifies the minimum number of digits to appear.

- For "e", "E" and "f" (fixed-precision, floating-point) conversions, the precision specifies the number of digits to appear after the decimal-point character.

- For "g" and "G" (variable-precision, floating-point) conversions, the precision specifies the maximum number of significant digits to appear.

- For "s" or "S" (string) conversions, the precision specifies the maximum number of characters to appear.

A type length specifier affects the conversion as follows:

- "h" causes a "d", "i", "o", "u", "x" or "X" (integer) format conversion to treat the argument as a `short int` or `unsigned short int` argument. Note that, although the argument may have been promoted to an `int` as part of the function call, the value is converted to the smaller type before it is formatted.

- "h" causes an "f" format conversion to interpret a `long` argument as a fixed-point number consisting of a 16-bit signed integer part and a 16-bit unsigned fractional part. The integer part is in the high 16 bits and the fractional part is in the low 16 bits.

```
struct fixpt {
    unsigned short fraction; /* Intel architecture! */
      signed short integral;
};

struct fixpt foo1 =
  { 0x8000, 1234 }; /* represents 1234.5 */
struct fixpt foo2 =
  { 0x8000, -1 };   /* represents -0.5 (-1+.5) */
```

The value is formatted with the same rules as for floating-point values. This is a Watcom extension.

- "h" causes an "n" (converted length assignment) operation to assign the converted length to an object of type `unsigned short int.`

- "h" causes an "s" operation to treat the argument string as an ASCII character string composed of 8-bit characters.

  For `printf` and related byte input/output functions, this specifier is redundant. For `wprintf` and related wide character input/output functions, this specifier is required if the argument string is to be treated as an 8-bit ASCII character string; otherwise it will be treated as a wide character string.

```
    printf(    "%s%d", "Num=", 12345 );
    wprintf( L"%hs%d", "Num=", 12345 );
```

- "l" causes a "d", "i", "o", "u", "x" or "X" (integer) conversion to process a `long int` or `unsigned long int` argument.

- "l" causes an "n" (converted length assignment) operation to assign the converted length to an object of type `unsigned long int.`

- "l" or "w" cause an "s" operation to treat the argument string as a wide character string (a string composed of characters of type `wchar_t`).

  For `printf` and related byte input/output functions, this specifier is required if the argument string is to be treated as a wide character string; otherwise it will be treated as an 8-bit ASCII character string. For `wprintf` and related wide character input/output functions, this specifier is redundant.

```
    printf(  "%ls%d", L"Num=", 12345 );
    wprintf( L"%s%d", L"Num=", 12345 );
```

- "F" causes the pointer associated with "n", "p", "s" conversions to be treated as a far pointer.

- "L" causes a "d", "i", "o", "u", "x" or "X" (integer) conversion to process an `__int64` or `unsigned __int64` argument (e.g., %Ld).

- "I64" causes a "d", "i", "o", "u", "x" or "X" (integer) conversion to process an `__int64` or `unsigned __int64` argument (e.g., %I64d). The "L" specifier provides the same functionality.

- "L" causes an "e", "E", "f", "g", "G" (double) conversion to process a `long double` argument.

- "N" causes the pointer associated with "n", "p", "s" conversions to be treated as a near pointer.

The valid conversion type specifiers are:

c An argument of type `int` is converted to a value of type `char` and the corresponding ASCII character code is written to the output stream.

C An argument of type `wchar_t` is converted to a multibyte character and written to the output stream.

d, i An argument of type `int` is converted to a signed decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

e, E An argument of type `double` is converted to a decimal notation in the form `[-]d.ddde[+|-]ddd` similar to FORTRAN exponential (E) notation. The leading sign appears (subject to the format control flags) only if the argument is negative. If the argument is non-zero, the digit before the decimal-point character is non-zero. The precision is used as the number of digits following the decimal-point character. If the precision is not specified, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed. The value is rounded to the appropriate number of digits. For "E" conversions, the exponent begins with the character "E" rather than "e". The exponent sign and a three-digit number (that indicates the power of ten by which the decimal fraction is multiplied) are always produced.

f An argument of type `double` is converted to a decimal notation in the form `[-]ddd.ddd` similar to FORTRAN fixed-point (F) notation. The leading sign appears (subject to the format control flags) only if the argument is negative. The precision is used as the number of digits following the decimal-point character. If the precision is not specified, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed, otherwise, at least one digit is produced before the decimal-point character. The value is rounded to the appropriate number of digits.

g, G An argument of type `double` is converted using either the "f" or "e" (or "E", for a "G" conversion) style of conversion depending on the value of the argument. In either case, the precision specifies the number of significant digits that are contained in the result. "e" style conversion is used only if the exponent from such a conversion would be less than -4 or greater than the precision. Trailing zeros are removed from the result and a decimal-point character only appears if it is followed by a digit.

n The number of characters that have been written to the output stream is assigned to the integer pointed to by the argument. No output is produced.

o An argument of type `int` is converted to an unsigned octal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

p, P An argument of type `void *` is converted to a value of type `int` and the value is formatted as for a hexadecimal ("x") conversion.

s Characters from the string specified by an argument of type `char *` or `wchar_t *`, up to, but not including the terminating null character ('\0'), are written to the output stream. If a precision is specified, no more than that many characters (bytes) are written (e.g., %.7s)

For `printf`, this specifier refers to an ASCII character string unless the "l" or "w" modifiers are used to indicate a wide character string.

For `wprintf`, this specifier refers to a wide character string unless the "h" modifier is used to indicate an ASCII character string.

S Characters from the string specified by an argument of type `wchar_t *`, up to, but not including the terminating null wide character (L'\0'), are converted to multibyte characters and written to the output stream. If a precision is specified, no more than that many characters (bytes) are written (e.g., %.7S)

u An argument of type `int` is converted to an unsigned decimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

x, X An argument of type `int` is converted to an unsigned hexadecimal notation and written to the output stream. The default precision is 1, but if more digits are required, leading zeros are added. Hexadecimal notation uses the digits "0" through "9" and the characters "a" through "f" or "A" through "F" for "x" or "X" conversions respectively, as the hexadecimal digits. Subject to the alternate-form control flag, "0x" or "0X" is prepended to the output.

Any other conversion type specifier character, including another percent character (%), is written to the output stream with no special interpretation.

The arguments must correspond with the conversion type specifiers, left to right in the string; otherwise, indeterminate results will occur.

If the value corresponding to a floating-point specifier is infinity, or not a number (NAN), then the output will be "inf" or "-inf" for infinity, and "nan" or "-nan" for NAN's.

For example, a specifier of the form `"%8.*f"` will define a field to be at least 8 characters wide, and will get the next argument for the precision to be used in the conversion.

Classification: ANSI, (except for F and N modifiers)

Systems:
```
printf - All, Netware
wprintf - All
```

Synopsis:

```
#include <stdio.h>
int putc( int c, FILE *fp );
#include <stdio.h>
#include <wchar.h>
wint_t putwc( wint_t c, FILE *fp );
```

Description: The `putc` function is equivalent to `fputc`, except it may be implemented as a macro. The `putc` function writes the character specified by the argument *c* to the output stream designated by *fp*.

The `putwc` function is identical to `putc` except that it converts the wide character specified by *c* to a multibyte character and writes it to the output stream.

Returns: The `putc` function returns the character written or, if a write error occurs, the error indicator is set and `putc` returns `EOF.`

The `putwc` function returns the wide character written or, if a write error occurs, the error indicator is set and `putwc` returns `WEOF.`

When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `fopen`, `fputc`, `fputchar`, `fputs`, `putchar`, `puts`, `ferror`

Example:

```
#include <stdio.h>

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      while( (c = fgetc( fp )) != EOF )
          putc( c, stdout );
      fclose( fp );
    }
  }
```

Classification: putc is ANSI, putwc is ANSI

Systems:

```
putc - All, Netware
putwc - All
```

Synopsis:
```
#include <conio.h>
int putch( int c );
```

Description: The `putch` function writes the character specified by the argument *c* to the console.

Returns: The `putch` function returns the character written.

See Also: `getch`, `getche`, `kbhit`, `ungetch`

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

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if ( fp != NULL ) {
      while( (c = fgetc( fp )) != EOF )
        putch( c );
    }
    fclose( fp );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <stdio.h>
int putchar( int c );
#include <wchar.h>
wint_t putwchar( wint_t c );
```

Description: The `putchar` function writes the character specified by the argument *c* to the output stream `stdout.`

The function is equivalent to

```
    fputc( c, stdout );
```

The `putwchar` function is identical to `putchar` except that it converts the wide character specified by *c* to a multibyte character and writes it to the output stream.

Returns: The `putchar` function returns the character written or, if a write error occurs, the error indicator is set and `putchar` returns `EOF.`

The `putwchar` function returns the wide character written or, if a write error occurs, the error indicator is set and `putwchar` returns `WEOF.`

When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `fopen`, `fputc`, `fputchar`, `fputs`, `putc`, `puts`, `ferror`

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

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    c = fgetc( fp );
    while( c != EOF ) {
        putchar( c );
        c = fgetc( fp );
    }
    fclose( fp );
  }
```

Classification: putchar is ANSI, putwchar is ANSI

Systems:
```
putchar - All, Netware
putwchar - All
```

Synopsis:
```
#include <process.h>
int putenv( const char *env_name );
int _putenv( const char *env_name );
int _wputenv( const wchar_t *env_name );
```

Description: The environment list consists of a number of environment names, each of which has a value associated with it. Entries can be added to the environment list with the DOS `set` command or with the `putenv` function. All entries in the environment list can be displayed by using the DOS `set` command with no arguments. A program can obtain the value for an environment variable by using the `getenv` function.

When the value of *env_name* has the format

```
        env_name=value
```

an environment name and its value is added to the environment list. When the value of *env_name* has the format

```
        env_name=
```

the environment name and value is removed from the environment list.

The matching is case-insensitive; all lowercase letters are treated as if they were in upper case.

The space into which environment names and their values are placed is limited. Consequently, the `putenv` function can fail when there is insufficient space remaining to store an additional value.

The `_putenv` function is identical to `putenv`. Use `_putenv` for ANSI naming conventions.

The `_wputenv` function is a wide-character version of `putenv` the *env_name* argument to `_wputenv` is a wide-character string.

`putenv` and `_wputenv` affect only the environment that is local to the current process; you cannot use them to modify the command-level environment. That is, these functions operate only on data structures accessible to the run-time library and not on the environment "segment" created for a process by the operating system. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level). However, the modified environment can be passed to any new processes created by _spawn, _exec, or system, and these new processes get any new items added by `putenv` and `_wputenv`.

With regard to environment entries, observe the following cautions:

- Do not change an environment entry directly; instead, use `putenv` or `_wputenv` to change it. To modify the return value of `putenv` or `_wputenv` without affecting the environment table, use `_strdup` or `strcpy` to make a copy of the string.

- If the argument *env_name* is not a literal string, you should duplicate the string, since `putenv` does not copy the value; for example,

```
putenv( _strdup( buffer ) );
```

- Never free a pointer to an environment entry, because the environment variable will then point to freed space. A similar problem can occur if you pass `putenv` or `_wputenv` a pointer to a local variable, then exit the function in which the variable is declared.

To assign a string to a variable and place it in the environment list:

```
C>SET INCLUDE=C:\WATCOM\H
```

To see what variables are in the environment list, and their current assignments:

```
C>SET
COMSPEC=C:\COMMAND.COM
PATH=C:\;C:\WATCOM
INCLUDE=C:\WATCOM\H

C>
```

Returns: The `putenv` function returns zero when it is successfully executed and returns -1 when it fails.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

ENOMEM	Not enough memory to allocate a new environment string.

See Also: `clearenv`, `getenv`, `setenv`

Example: The following gets the string currently assigned to INCLUDE and displays it, assigns a new value to it, gets and displays it, and then removes the environment name and value.

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

void main()
  {
    char *path;
    path = getenv( "INCLUDE" );
    if( path != NULL )
        printf( "INCLUDE=%s\n", path );
    if( putenv( "INCLUDE=mylib;yourlib" ) != 0 )
        printf( "putenv failed" );
    path = getenv( "INCLUDE" );
    if( path != NULL )
        printf( "INCLUDE=%s\n", path );
    if( putenv( "INCLUDE=" ) != 0 )
        printf( "putenv failed" );
  }
```

produces the following:

```
INCLUDE=C:\WATCOM\H
INCLUDE=mylib;yourlib
```

Classification: putenv is POSIX 1003.1, _putenv is not POSIX, _wputenv is not POSIX

Systems:
```
putenv - All
_putenv - All
_wputenv - All
```

Synopsis:

```
#include <graph.h>
void _FAR _putimage( short x, short y,
                     char _HUGE *image, short mode );

void _FAR _putimage_w( double x, double y,
                       char _HUGE *image, short mode );
```

Description: The `_putimage` functions display the screen image indicated by the argument *image*. The `_putimage` function uses the view coordinate system. The `_putimage_w` function uses the window coordinate system.

The image is displayed upon the screen with its top left corner located at the point with coordinates `(x,y).` The image was previously saved using the `_getimage` functions. The image is displayed in a rectangle whose size is the size of the rectangular image saved by the `_getimage` functions.

The image can be displayed in a number of ways, depending upon the value of the *mode* argument. This argument can have the following values:

_GPSET	replace the rectangle on the screen by the saved image
_GPRESET	replace the rectangle on the screen with the pixel values of the saved image inverted; this produces a negative image
_GAND	produce a new image on the screen by ANDing together the pixel values from the screen with those from the saved image
_GOR	produce a new image on the screen by ORing together the pixel values from the screen with those from the saved image
_GXOR	produce a new image on the screen by exclusive ORing together the pixel values from the screen with those from the saved image; the original screen is restored by two successive calls to the `_putimage` function with this value, providing an efficient method to produce animated effects

Returns: The `_putimage` functions do not return a value.

See Also: `_getimage`, `_imagesize`

Example:

```
#include <conio.h>
#include <graph.h>
#include <malloc.h>

main()
{
    char *buf;
    int y;

    _setvideomode( _VRES16COLOR );
    _ellipse( _GFILLINTERIOR, 100, 100, 200, 200 );
    buf = (char*) malloc(
                  _imagesize( 100, 100, 201, 201 ) );
    if( buf != NULL ) {
        _getimage( 100, 100, 201, 201, buf );
        _putimage( 260, 200, buf, _GPSET );
        _putimage( 420, 100, buf, _GPSET );
        for( y = 100; y < 300; ) {
            _putimage( 420, y, buf, _GXOR );
            y += 20;
            _putimage( 420, y, buf, _GXOR );
        }
        free( buf );
    }
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: _putimage is PC Graphics

Systems:

```
_putimage - DOS, QNX
_putimage_w - DOS, QNX
```

Synopsis:

```
#include <stdio.h>
int puts( const char *buf );
#include <stdio.h>
int _putws( const wchar_t *bufs );
```

Description: The `puts` function writes the character string pointed to by *buf* to the output stream designated by `stdout`, and appends a new-line character to the output. The terminating null character is not written.

The `_putws` function is identical to `puts` except that it converts the wide character string specified by *buf* to a multibyte character string and writes it to the output stream.

Returns: The `puts` function returns `EOF` if an error occurs; otherwise, it returns a non-negative value (the amount written including the new-line character). The `_putws` function returns `WEOF` if a write or encoding error occurs; otherwise, it returns a non-negative value (the amount written including the new-line character). When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `fopen`, `fputc`, `fputchar`, `fputs`, `putc`, `putchar`, `ferror`

Example:

```
#include <stdio.h>

void main()
  {
    FILE *fp;
    char buffer[80];

    fp = freopen( "file", "r", stdin );
    while( gets( buffer ) != NULL ) {
        puts( buffer );
    }
    fclose( fp );
  }
```

Classification: puts is ANSI, _putws is not ANSI

Systems:

```
puts - All, Netware
_putws - All
```

Synopsis:
```
#include <stdio.h>
int _putw( int binint, FILE *fp );
```

Description: The `_putw` function writes a binary value of type *int* to the current position of the stream *fp*. `_putw` does not affect the alignment of items in the stream, nor does it assume any special alignment.

`_putw` is provided primarily for compatibility with previous libraries. Portability problems may occur with `_putw` because the size of an *int* and the ordering of bytes within an *int* differ across systems.

Returns: The `_putw` function returns the value written or, if a write error occurs, the error indicator is set and `_putw` returns `EOF.` Since `EOF` is a legitimate value to write to *fp*, use `ferror` to verify that an error has occurred.

See Also: `ferror`, `fopen`, `fputc`, `fputchar`, `fputs`, `putc`, `putchar`, `puts`

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

void main()
  {
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
      while( (c = _getw( fp )) != EOF )
          _putw( c, stdout );
      fclose( fp );
    }
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
void qsort( void *base,
            size_t num,
            size_t width,
            int (*compar)
                 ( const void *, const void *) );
```

Description: The `qsort` function sorts an array of *num* elements, which is pointed to by *base*, using a modified version of Sedgewick's Quicksort algorithm. Each element in the array is *width* bytes in size. The comparison function pointed to by *compar* is called with two arguments that point to elements in the array. The comparison function shall return an integer less than, equal to, or greater than zero if the first argument is less than, equal to, or greater than the second argument.

The version of the Quicksort algorithm that is employed was proposed by Jon Louis Bentley and M. Douglas McIlroy in the article "Engineering a sort function" published in ***Software -- Practice and Experience,*** 23(11):1249-1265, November 1993.

Returns: The `qsort` function returns no value.

See Also: `bsearch`

Example:

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

char *CharVect[] = { "last", "middle", "first" };

int compare( const void *op1, const void *op2 )
  {
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;
    return( strcmp( *p1, *p2 ) );
  }

void main()
  {
    qsort( CharVect, sizeof(CharVect)/sizeof(char *),
          sizeof(char *), compare );
    printf( "%s %s %s\n",
            CharVect[0], CharVect[1], CharVect[2] );
  }
```

produces the following:

```
first last middle
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <signal.h>
int raise( int condition );
```

Description: The `raise` function signals the exceptional condition indicated by the *condition* argument. The possible conditions are defined in the `<signal.h>` header file and are documented with the `signal` function. The `signal` function can be used to specify the action which is to take place when such a condition occurs.

Returns: The `raise` function returns zero when the condition is successfully raised and a non-zero value otherwise. There may be no return of control following the function call if the action for that condition is to terminate the program or to transfer control using the `longjmp` function.

See Also: `signal`

Example:
```
/*
 * This program waits until a SIGINT signal
 * is received.
 */
#include <stdio.h>
#include <signal.h>

sig_atomic_t signal_count;
sig_atomic_t signal_number;

static void alarm_handler( int signum )
  {
    ++signal_count;
    signal_number = signum;
  }

void main()
  {
    unsigned long i;

    signal_count = 0;
    signal_number = 0;
    signal( SIGINT, alarm_handler );
```

```
        printf("Signal will be auto-raised on iteration "
               "10000 or hit CTRL-C.\n");
        printf("Iteration:      ");
        for( i = 0; i < 100000; ++i )
        {
          printf("\b\b\b\b\b%*d", 5, i);

          if( i == 10000 ) raise(SIGINT);

          if( signal_count > 0 ) break;
        }

        if( i == 100000 ) {
          printf("\nNo signal was raised.\n");
        } else if( i == 10000 ) {
          printf("\nSignal %d was raised by the "
                  "raise() function.\n", signal_number);
        } else {
          printf("\nUser raised the signal.\n",
                  signal_number);
        }
      }
```

Classification: ANSI

Systems: All, Netware

Synopsis:

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

Description: The `rand` function computes a sequence of pseudo-random integers in the range 0 to `RAND_MAX` (32767). The sequence can be started at different values by calling the `srand` function.

Returns: The `rand` function returns a pseudo-random integer.

See Also: `srand`

Example:

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

void main()
  {
    int i;

    for( i=1; i < 10; ++i ) {
      printf( "%d\n", rand() );
    }
  }
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <io.h>
int read( int handle, void *buffer, unsigned len );
```

Description: The `read` function reads data at the operating system level. The number of bytes transmitted is given by *len* and the data is transmitted starting at the address specified by *buffer*.

The *handle* value is returned by the `open` function. The access mode must have included either `O_RDONLY` or `O_RDWR` when the `open` function was invoked. The data is read starting at the current file position for the file in question. This file position can be determined with the `tell` function and can be set with the `lseek` function.

When `O_BINARY` is included in the access mode, the data is transmitted unchanged. When `O_TEXT` is included in the access mode, the data is transmitted with the extra carriage return character removed before each linefeed character encountered in the original data.

Returns: The `read` function returns the number of bytes of data transmitted from the file to the buffer (this does not include any carriage-return characters that were removed during the transmission). Normally, this is the number given by the *len* argument. When the end of the file is encountered before the read completes, the return value will be less than the number of bytes requested.

A value of -1 is returned when an input/output error is detected. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `close`, `creat`, `fread`, `open`, `write`

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

void main()
  {
    int  handle;
    int  size_read;
    char buffer[80];

    /* open a file for input              */
    handle = open( "file", O_RDONLY | O_TEXT );
    if( handle != -1 ) {
      /* read the text                      */
      size_read = read( handle, buffer,
                        sizeof( buffer ) );

      /* test for error                     */
      if( size_read == -1 ) {
          printf( "Error reading file\n" );
      }

      /* close the file                     */
      close( handle );
    }
  }
```

Classification: POSIX 1003.1

Systems: All, Netware

Synopsis:

```
#include <direct.h>
struct dirent *readdir( struct dirent *dirp );
struct _wdirent *_wreaddir( _wdirent *dirp );
```

Description: The `readdir` function obtains information about the next matching file name from the argument *dirp*. The argument *dirp* is the value returned from the `opendir` function. The `readdir` function can be called repeatedly to obtain the list of file names contained in the directory specified by the pathname given to `opendir.` The function `closedir` must be called to close the directory and free the memory allocated by `opendir.`

The file `<direct.h>` contains definitions for the structure `dirent.`

```
#if defined(__OS2__) || defined(__NT__)
#define NAME_MAX 255    /* maximum for HPFS or NTFS */
#else
#define NAME_MAX  12    /* 8 chars + '.' +  3 chars */
#endif

typedef struct dirent {
    char    d_dta[ 21 ];        /* disk transfer area */
    char    d_attr;             /* file's attribute */
    unsigned short int d_time;  /* file's time */
    unsigned short int d_date;  /* file's date */
    long    d_size;             /* file's size */
    char    d_name[ NAME_MAX + 1 ]; /* file's name */
    unsigned short d_ino;       /* serial number */
    char    d_first;            /* flag for 1st time */
} DIR;
```

The file attribute field `d_attr` field is a set of bits representing the following attributes.

```
_A_RDONLY       /* Read-only file */
_A_HIDDEN       /* Hidden file */
_A_SYSTEM       /* System file */
_A_VOLID        /* Volume-ID entry (only MSFT knows) */
_A_SUBDIR       /* Subdirectory */
_A_ARCH         /* Archive file */
```

If the `_A_RDONLY` bit is off, then the file is read/write.

The format of the `d_time` field is described by the following structure (this structure is not defined in any Watcom header file).

```
typedef struct {
    unsigned short  twosecs : 5;    /* seconds / 2 */
    unsigned short  minutes : 6;    /* minutes (0,59) */
    unsigned short  hours   : 5;    /* hours (0,23) */
} ftime_t;
```

The format of the `d_date` field is described by the following structure (this structure is not defined in any Watcom header file).

```
typedef struct {
    unsigned short  day     : 5;    /* day (1,31) */
    unsigned short  month   : 4;    /* month (1,12) */
    unsigned short  year    : 7;    /* 0 is 1980 */
} fdate_t;
```

See the sample program below for an example of the use of these structures.

The `_wreaddir` function is identical to `readdir` except that it reads a directory of wide-character filenames.

The file `<direct.h>` contains definitions for the structure `_wdirent.`

```
struct _wdirent {
    char    d_dta[21];      /* disk transfer area */
    char    d_attr;         /* file's attribute */
    unsigned short int d_time;/* file's time */
    unsigned short int d_date;/* file's date */
    long    d_size;         /* file's size */
    wchar_t d_name[NAME_MAX+1];/* file's name */
    unsigned short d_ino;   /* serial number (not used) */
    char    d_first;        /* flag for 1st time */
};
```

Returns: When successful, `readdir` returns a pointer to an object of type *struct dirent.* When an error occurs, `readdir` returns the value `NULL` and `errno` is set to indicate the error. When the end of the directory is encountered, `readdir` returns the value `NULL` and `errno` is unchanged.

When successful, `_wreaddir` returns a pointer to an object of type *struct _wdirent.* When an error occurs, `_wreaddir` returns the value `NULL` and `errno` is set to indicate the error. When the end of the directory is encountered, `_wreaddir` returns the value `NULL` and `errno` is unchanged.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

EBADF	The argument *dirp* does not refer to an open directory stream.

See Also: `closedir`, `_dos_find` Functions, `opendir`, `rewinddir`

Example: To get a list of files contained in the directory `\watcom\h` on your default disk:

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

typedef struct {
    unsigned short  twosecs : 5;    /* seconds / 2 */
    unsigned short  minutes : 6;
    unsigned short  hours   : 5;
} ftime_t;

typedef struct {
    unsigned short  day     : 5;
    unsigned short  month   : 4;
    unsigned short  year    : 7;
} fdate_t;

void main()
  {
    DIR *dirp;
    struct dirent *direntp;
    ftime_t *f_time;
    fdate_t *f_date;
```

```
      dirp = opendir( "\\watcom\\h" );
      if( dirp != NULL ) {
        for(;;) {
          direntp = readdir( dirp );
          if( direntp == NULL ) break;
          f_time = (ftime_t *)&direntp->d_time;
          f_date = (fdate_t *)&direntp->d_date;
          printf( "%-12s %d/%2.2d/%2.2d "
                  "%2.2d:%2.2d:%2.2d \n",
              direntp->d_name,
              f_date->year + 1980,
              f_date->month,
              f_date->day,
              f_time->hours,
              f_time->minutes,
              f_time->twosecs * 2 );
        }
        closedir( dirp );
      }
    }
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: readdir is POSIX 1003.1, _wreaddir is not POSIX

Systems:

```
readdir - All, Netware
_wreaddir - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <stdlib.h>  For ANSI compatibility (realloc only)
#include <malloc.h>  Required for other function prototypes
void * realloc( void *old_blk, size_t size );
void __based(void) *_brealloc( __segment seg,
                               void __based(void) *old_blk,
                               size_t size );
void __far  *_frealloc( void __far  *old_blk,
                              size_t size );
void __near *_nrealloc( void __near *old_blk,
                              size_t size );
```

Description: When the value of the *old_blk* argument is `NULL`, a new block of memory of *size* bytes is allocated.

If the value of *size* is zero, the corresponding `free` function is called to release the memory pointed to by *old_blk.*

Otherwise, the `realloc` function re-allocates space for an object of *size* bytes by either:

- shrinking the allocated size of the allocated memory block *old_blk* when *size* is sufficiently smaller than the size of *old_blk.*
- extending the allocated size of the allocated memory block *old_blk* if there is a large enough block of unallocated memory immediately following *old_blk.*
- allocating a new block and copying the contents of *old_blk* to the new block.

Because it is possible that a new block will be allocated, any pointers into the old memory should not be maintained. These pointers will point to freed memory, with possible disastrous results, when a new block is allocated.

The function returns `NULL` when the memory pointed to by *old_blk* cannot be re-allocated. In this case, the memory pointed to by *old_blk* is not freed so care should be exercised to maintain a pointer to the old memory block.

```
buffer = (char *) realloc( buffer, 100 );
```

In the above example, `buffer` will be set to `NULL` if the function fails and will no longer point to the old memory block. If `buffer` was your only pointer to the memory block then you will have lost access to this memory.

Each function reallocates memory from a particular heap, as listed below:

Function	***Heap***
realloc	Depends on data model of the program
_brealloc	Based heap specified by *seg* value
_frealloc	Far heap (outside the default data segment)
_nrealloc	Near heap (inside the default data segment)

In a small data memory model, the `realloc` function is equivalent to the `_nrealloc` function; in a large data memory model, the `realloc` function is equivalent to the `_frealloc` function.

Returns: The `realloc` functions return a pointer to the start of the re-allocated memory. The return value is `NULL` if there is insufficient memory available or if the value of the *size* argument is zero. The `_brealloc` function returns `_NULLOFF` if there is insufficient memory available or if the requested size is zero.

See Also: `calloc` Functions, `_expand` Functions, `free` Functions, `halloc`, `hfree`, `malloc` Functions, `_msize` Functions, `sbrk`

Example:

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

void main()
  {
    char *buffer;
    char *new_buffer;

    buffer = (char *) malloc( 80 );
    new_buffer = (char *) realloc( buffer, 100 );
    if( new_buffer == NULL ) {

      /* not able to allocate larger buffer */

    } else {
      buffer = new_buffer;
    }
  }
```

Classification: realloc is ANSI, _frealloc is not ANSI, _brealloc is not ANSI, _nrealloc is not ANSI

Systems: `realloc - All, Netware`

```
_brealloc - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_frealloc - DOS/16, Windows, QNX/16, OS/2 1.x(all)
_nrealloc - DOS, Windows, Win386, Win32, QNX, OS/2 1.x, OS/2
1.x(MT), OS/2-32
```

Synopsis:

```
#include <graph.h>
short _FAR _rectangle( short fill,
                       short x1, short y1,
                       short x2, short y2 );

short _FAR _rectangle_w( short fill,
                         double x1, double y1,
                         double x2, double y2 );

short _FAR _rectangle_wxy( short fill,
                           struct _wxycoord _FAR *p1,
                           struct _wxycoord _FAR *p2 );
```

Description: The `_rectangle` functions draw rectangles. The `_rectangle` function uses the view coordinate system. The `_rectangle_w` and `_rectangle_wxy` functions use the window coordinate system.

The rectangle is defined with opposite corners established by the points `(x1,y1)` and `(x2,y2)`.

The argument *fill* determines whether the rectangle is filled in or has only its outline drawn. The argument can have one of two values:

_GFILLINTERIOR	fill the interior by writing pixels with the current plot action using the current color and the current fill mask
_GBORDER	leave the interior unchanged; draw the outline of the figure with the current plot action using the current color and line style

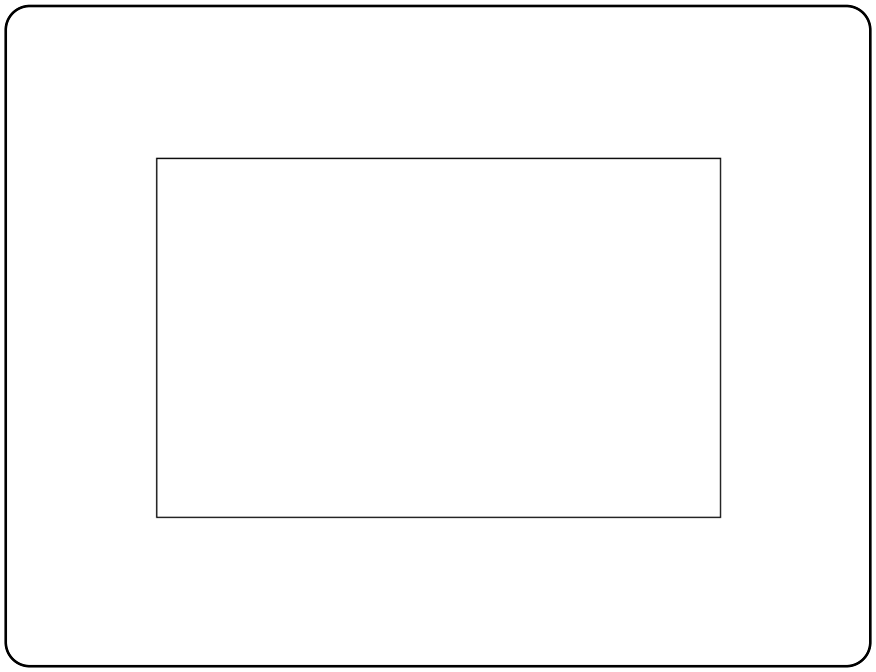
Returns: The `_rectangle` functions return a non-zero value when the rectangle was successfully drawn; otherwise, zero is returned.

See Also: `_setcolor`, `_setfillmask`, `_setlinestyle`, `_setplotaction`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _rectangle( _GBORDER, 100, 100, 540, 380 );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: _rectangle is PC Graphics

Systems:
```
_rectangle - DOS, QNX
_rectangle_w - DOS, QNX
_rectangle_wxy - DOS, QNX
```

Synopsis:
```
#include <graph.h>
short _FAR _registerfonts( char _FAR *path );
```

Description: The `_registerfonts` function initializes the font graphics system. Fonts must be registered, and a font selected, before text can be displayed with the `_outgtext` function.

The argument *path* specifies the location of the font files. This argument is a file specification, and can contain drive and directory components and may contain wildcard characters. The `_registerfonts` function opens each of the font files specified and reads the font information. Memory is allocated to store the characteristics of the font. These font characteristics are used by the `_setfont` function when selecting a font.

Returns: The `_registerfonts` function returns the number of fonts that were registered if the function is successful; otherwise, a negative number is returned.

See Also: `_unregisterfonts`, `_setfont`, `_getfontinfo`, `_outgtext`, `_getgtextextent`, `_setgtextvector`, `_getgtextvector`

Example:
```
#include <conio.h>
#include <stdio.h>
#include <graph.h>

main()
{
    int i, n;
    char buf[ 10 ];

    _setvideomode( _VRES16COLOR );
    n = _registerfonts( "*.fon" );
    for( i = 0; i < n; ++i ) {
        sprintf( buf, "n%d", i );
        _setfont( buf );
        _moveto( 100, 100 );
        _outgtext( "WATCOM Graphics" );
        getch();
        _clearscreen( _GCLEARSCREEN );
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _remapallpalette( long _FAR *colors );
```

Description: The `_remapallpalette` function sets (or remaps) all of the colors in the palette. The color values in the palette are replaced by the array of color values given by the argument *colors.* This function is supported in all video modes, but only works with EGA, MCGA and VGA adapters.

The array *colors* must contain at least as many elements as there are supported colors. The newly mapped palette will cause the complete screen to change color wherever there is a pixel value of a changed color in the palette.

The representation of colors depends upon the hardware being used. The number of colors in the palette can be determined by using the `_getvideoconfig` function.

Returns: The `_remapallpalette` function returns (-1) if the palette is remapped successfully and zero otherwise.

See Also: `_remappalette`, `_getvideoconfig`

Example:

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

long colors[ 16 ] = {
   _BRIGHTWHITE, _YELLOW, _LIGHTMAGENTA, _LIGHTRED,
   _LIGHTCYAN, _LIGHTGREEN, _LIGHTBLUE, _GRAY, _WHITE,
   _BROWN, _MAGENTA, _RED, _CYAN, _GREEN, _BLUE, _BLACK,
};

main()
{
    int x, y;

    _setvideomode( _VRES16COLOR );
    for( y = 0; y < 4; ++y ) {
        for( x = 0; x < 4; ++x ) {
            _setcolor( x + 4 * y );
            _rectangle( _GFILLINTERIOR,
                    x * 160, y * 120,
                    ( x + 1 ) * 160, ( y + 1 ) * 120 );
        }
    }
    getch();
    _remapallpalette( colors );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
long _FAR _remappalette( short pixval, long color );
```

Description: The `_remappalette` function sets (or remaps) the palette color *pixval* to be the color *color.* This function is supported in all video modes, but only works with EGA, MCGA and VGA adapters.

The argument *pixval* is an index in the color palette of the current video mode. The argument *color* specifies the actual color displayed on the screen by pixels with pixel value *pixval.* Color values are selected by specifying the red, green and blue intensities that make up the color. Each intensity can be in the range from 0 to 63, resulting in 262144 possible different colors. A given color value can be conveniently specified as a value of type `long.` The color value is of the form `0x00bbggrr,` where `bb` is the blue intensity, `gg` is the green intensity and `rr` is the red intensity of the selected color. The file `graph.h` defines constants containing the color intensities of each of the 16 default colors.

The `_remappalette` function takes effect immediately. All pixels on the complete screen which have a pixel value equal to the value of *pixval* will now have the color indicated by the argument *color.*

Returns: The `_remappalette` function returns the previous color for the pixel value if the palette is remapped successfully; otherwise, (-1) is returned.

See Also: `_remapallpalette`, `_setvideomode`

Example:

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

long colors[ 16 ] = {
   _BLACK, _BLUE, _GREEN, _CYAN,
   _RED, _MAGENTA, _BROWN, _WHITE,
   _GRAY, _LIGHTBLUE, _LIGHTGREEN, _LIGHTCYAN,
   _LIGHTRED, _LIGHTMAGENTA, _YELLOW, _BRIGHTWHITE
};

main()
{
    int col;

    _setvideomode( _VRES16COLOR );
    for( col = 0; col < 16; ++col ) {
        _remappalette( 0, colors[ col ] );
        getch();
    }
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <stdio.h>
int remove( const char *filename );
int _wremove( const wchar_t *filename );
```

Description: The `remove` function deletes the file whose name is the string pointed to by *filename*.

The `_wremove` function is identical to `remove` except that it accepts a wide-character string argument.

Returns: The `remove` function returns zero if the operation succeeds, non-zero if it fails. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Example:

```
#include <stdio.h>

void main()
  {
    remove( "vm.tmp" );
  }
```

Classification: remove is ANSI, _wremove is not ANSI

Systems:

```
remove - All, Netware
_wremove - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <stdio.h>
int rename( const char *old, const char *new );
int _wrename( const wchar_t *old, const wchar_t *new );
```

Description: The `rename` function causes the file whose name is indicated by the string *old* to be renamed to the name given by the string *new*. The `_wrename` function is identical to `rename` except that it accepts wide-character string arguments.

Returns: The `rename` function returns zero if the operation succeeds, a non-zero value if it fails. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

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

void main()
  {
    rename( "old.dat", "new.dat" );
  }
```

Classification: rename is ANSI, _wrename is not ANSI

Systems:
```
rename - All, Netware
_wrename - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <stdio.h>
void rewind( FILE *fp );
```

Description: The `rewind` function sets the file position indicator for the stream indicated to by *fp* to the beginning of the file. It is equivalent to

```
        fseek( fp, 0L, SEEK_SET );
```

except that the error indicator for the stream is cleared.

Returns: The `rewind` function returns no value.

See Also: `fopen`, `clearerr`

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

static assemble_pass( int passno )
  {
    printf( "Pass %d\n", passno );
  }

void main()
  {
    FILE *fp;

    if( (fp = fopen( "program.asm", "r")) != NULL ) {
        assemble_pass( 1 );
        rewind( fp );
        assemble_pass( 2 );
        fclose( fp );
    }
  }
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <sys\types.h>
#include <direct.h>
void rewinddir( struct dirent *dirp );
void _wrewinddir( _wdirent *dirp );
```

Description: The `rewinddir` function resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to `opendir` would have done.

The `_wrewinddir` function is identical to `rewinddir` except that it rewinds a directory of wide-character filenames opened by `_wopendir.`

Returns: The `rewinddir` function does not return a value.

See Also: `closedir`, `_dos_find` Functions, `opendir`, `readdir`

Example: The following example lists all the files in a directory, creates a new file, and then relists the directory.

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

void main()
  {
    DIR *dirp;
    struct dirent *direntp;
    int handle;

    dirp = opendir( "\\watcom\\h\\*.*" );
    if( dirp != NULL ) {
      printf( "Old directory listing\n" );
      for(;;) {
          direntp = readdir( dirp );
          if( direntp == NULL ) break;
          printf( "%s\n", direntp->d_name );
      }

      handle = creat( "\\watcom\\h\\file.new",
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
      close( handle );
```

```
        rewinddir( dirp );
        printf( "New directory listing\n" );
        for(;;) {
          direntp = readdir( dirp );
          if( direntp == NULL ) break;
          printf( "%s\n", direntp->d_name );
        }
        closedir( dirp );
      }
    }
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: rewinddir is POSIX 1003.1, _wrewinddir is not POSIX

Systems:
```
rewinddir - All
_wrewinddir - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <sys\types.h>
#include <direct.h>
int rmdir( const char *path );
int _wrmdir( const wchar_t *path );
```

Description: The `rmdir` function removes (deletes) the specified directory. The directory must not contain any files or directories. The *path* can be either relative to the current working directory or it can be an absolute path name.

The `_wrmdir` function is identical to `rmdir` except that it accepts a wide-character string argument.

Returns: The `rmdir` function returns zero if successful and -1 otherwise.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `chdir`, `chmod`, `getcwd`, `mkdir`, `stat`, `umask`

Example: To remove the directory called `\watcom` on drive `C:`

```
#include <sys\types.h>
#include <direct.h>

void main()
  {
    rmdir( "c:\\watcom" );
  }
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: rmdir is POSIX 1003.1, _wrmdir is not POSIX

Systems:

```
rmdir - All, Netware
_wrmdir - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <stdlib.h>
unsigned int _rotl( unsigned int value,
                    unsigned int shift );
```

Description: The `_rotl` function rotates the unsigned integer, determined by *value*, to the left by the number of bits specified in *shift*. If you port an application using `_rotl` between a 16-bit and a 32-bit environment, you will get different results because of the difference in the size of integers.

Returns: The rotated value is returned.

See Also: `_lrotl`, `_lrotr`, `_rotr`

Example:

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

unsigned int mask = 0x0F00;

void main()
  {
    mask = _rotl( mask, 4 );
    printf( "%04X\n", mask );
  }
```

produces the following:

```
F000
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
unsigned int _rotr( unsigned int value,
                    unsigned int shift );
```

Description: The `_rotr` function rotates the unsigned integer, determined by *value*, to the right by the number of bits specified in *shift*. If you port an application using `_rotr` between a 16-bit and a 32-bit environment, you will get different results because of the difference in the size of integers.

Returns: The rotated value is returned.

See Also: `_lrotl`, `_lrotr`, `_rotl`

Example:

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

unsigned int mask = 0x1230;

void main()
  {
    mask = _rotr( mask, 4 );
    printf( "%04X\n", mask );
  }
```

produces the following:

```
0123
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
void *sbrk( int increment );
```

Description: Under 16-bit DOS and Phar Lap's 386|DOS-Extender, the data segment is grown contiguously. The "break" value is the address of the first byte of unallocated memory. When a program starts execution, the break value is placed following the code and constant data for the program. As memory is allocated, this pointer will advance when there is no freed block large enough to satisfy an allocation request. The `sbrk` function can be used to set a new "break" value for the program by adding the value of *increment* to the current break value. This increment may be positive or negative.

Under other systems, heap allocation is discontiguous. The `sbrk` function can only be used to allocate additional discontiguous blocks of memory. The value of *increment* is used to determine the minimum size of the block to be allocated and may not be zero or negative. The actual size of the block that is allocated is rounded up to a multiple of 4K.

The variable `_amblksiz` defined in `<stdlib.h>` contains the default increment by which the "break" pointer for memory allocation will be advanced when there is no freed block large enough to satisfy a request to allocate a block of memory. This value may be changed by a program at any time.

Under 16-bit DOS, a new process started with one of the `spawn...` or `exec...` functions is loaded following the break value. Consequently, decreasing the break value leaves more space available to the new process. Similarly, for a resident program (a program which remains in memory while another program executes), increasing the break value will leave more space available to be allocated by the resident program after other programs are loaded.

Returns: If the call to `sbrk` succeeds, a pointer to the start of the new block of memory is returned. Under 16-bit DOS, this corresponds to the old break value. If the call to `sbrk` fails, -1 is returned. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `calloc` Functions, `_expand` Functions, `free` Functions, `halloc`, `hfree`, `malloc` Functions, `_msize` Functions, `realloc` Functions

Example:

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

#if defined(M_I86)
#define alloc( x, y ) sbrk( x ); y = sbrk( 0 );
#else
#define alloc( x, y ) y = sbrk( x );
#endif

void main()
 {
    void *brk;

#if defined(M_I86)
    alloc( 0x0000, brk );
    /* calling printf will cause an allocation */
    printf( "Original break value %p\n", brk );
    printf( "Current amblksiz value %x\n", _amblksiz );
    alloc( 0x0000, brk );
    printf( "New break value after printf \t\t%p\n", brk );
#endif
    alloc( 0x3100, brk );
    printf( "New break value after sbrk( 0x3100 ) \t%p\n",
            brk );
    alloc( 0x0200, brk );
    printf( "New break value after sbrk( 0x0200 ) \t%p\n",
            brk );
#if defined(M_I86)
    alloc( -0x0100, brk );
    printf( "New break value after sbrk( -0x0100 ) \t%p\n",
            brk );
#endif
 }
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, QNX, OS/2 1.x, OS/2 1.x(MT), OS/2-32

Synopsis:

```
#include <stdio.h>
int scanf( const char *format, ... );
#include <wchar.h>
int wscanf( const wchar_t *format, ... );
```

Description: The `scanf` function scans input from the file designated by `stdin` under control of the argument *format.* The *format* string is described below. Following the format string is the list of addresses of items to receive values.

The `wscanf` function is identical to `scanf` except that it accepts a wide-character string argument for *format.*

Returns: The `scanf` function returns `EOF` when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

See Also: `cscanf`, `fscanf`, `sscanf`, `vcscanf`, `vfscanf`, `vscanf`, `vsscanf`

Example: To scan a date in the form "Saturday April 18 1987":

```
#include <stdio.h>

void main()
  {
    int day, year;
    char weekday[10], month[10];

    scanf( "%s %s %d %d", weekday, month, &day, &year );
  }
```

Format Control String: The format control string consists of zero or more *format directives* that specify acceptable input file data. Subsequent arguments are pointers to various types of objects that are assigned values as the format string is processed.

A format directive can be a sequence of one or more white-space characters, an *ordinary character,* or a *conversion specifier.* An ordinary character in the format string is any character, other than a white-space character or the percent character (%), that is not part of a conversion specifier. A conversion specifier is a sequence of characters in the format string that begins with a percent character (%) and is followed, in sequence, by the following:

- an optional assignment suppression indicator: the asterisk character (*);
- an optional decimal integer that specifies the *maximum field width* to be scanned for the conversion;

- an optional *pointer-type* specification: one of "N" or "F";
- an optional *type length* specification: one of "h", "l", "L" or "I64";
- a character that specifies the type of conversion to be performed: one of the characters "cCdefginopsSux[".

As each format directive in the format string is processed, the directive may successfully complete, fail because of a lack of input data, or fail because of a matching error as defined by the particular directive. If end-of-file is encountered on the input data before any characters that match the current directive have been processed (other than leading white-space where permitted), the directive fails for lack of data. If end-of-file occurs after a matching character has been processed, the directive is completed (unless a matching error occurs), and the function returns without processing the next directive. If a directive fails because of an input character mismatch, the character is left unread in the input stream. Trailing white-space characters, including new-line characters, are not read unless matched by a directive. When a format directive fails, or the end of the format string is encountered, the scanning is completed and the function returns.

When one or more white-space characters (space " ", horizontal tab "\t", vertical tab "\v", form feed "\f", carriage return "\r", new line or linefeed "\n") occur in the format string, input data up to the first non-white-space character is read, or until no more data remains. If no white-space characters are found in the input data, the scanning is complete and the function returns.

An ordinary character in the format string is expected to match the same character in the input stream.

A conversion specifier in the format string is processed as follows:

- for conversion types other than "[", "c", "C" and "n", leading white-space characters are skipped
- for conversion types other than "n", all input characters, up to any specified maximum field length, that can be matched by the conversion type are read and converted to the appropriate type of value; the character immediately following the last character to be matched is left unread; if no characters are matched, the format directive fails
- unless the assignment suppression indicator ("*") was specified, the result of the conversion is assigned to the object pointed to by the next unused argument (if assignment suppression was specified, no argument is skipped); the arguments must correspond in number, type and order to the conversion specifiers in the format string

A pointer-type specification is used to indicate the type of pointer used to locate the next argument to be scanned:

F pointer is a far pointer

N pointer is a near pointer

The pointer type defaults to that used for data in the memory model for which the program has been compiled.

A type length specifier affects the conversion as follows:

- "h" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type `short int` or `unsigned short int.`

- "h" causes an "f" conversion to assign a fixed-point number to an object of type `long` consisting of a 16-bit signed integer part and a 16-bit unsigned fractional part. The integer part is in the high 16 bits and the fractional part is in the low 16 bits.

```
struct fixpt {
    unsigned short fraction; /* Intel architecture! */
      signed short integral;
};

struct fixpt foo1 =
  { 0x8000, 1234 }; /* represents 1234.5 */
struct fixpt foo2 =
  { 0x8000, -1 };   /* represents -0.5 (-1+.5) */
```

- "h" causes an "n" (read length assignment) operation to assign the number of characters that have been read to an object of type `unsigned short int.`

- "h" causes an "s" operation to convert the input string to an ASCII character string. For `scanf`, this specifier is redundant. For `wscanf`, this specifier is required if the wide character input string is to be converted to an ASCII character string; otherwise it will not be converted.

- "l" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type `long int` or `unsigned long int.`

- "l" causes an "n" (read length assignment) operation to assign the number of characters that have been read to an object of type `unsigned long int.`

- "l" causes an "e", "f" or "g" (floating-point) conversion to assign the converted value to an object of type `double.`

- "l" or "w" cause an "s" operation to convert the input string to a wide character string. For `scanf`, this specifier is required if the input ASCII string is to be converted to a wide character string; otherwise it will not be converted.

- "L" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type `__int64` or `unsigned __int64` (e.g., %Ld).

- "I64" causes a "d", "i", "o", "u" or "x" (integer) conversion to assign the converted value to an object of type `__int64` or `unsigned __int64` (e.g., %I64d). The "L" specifier provides the same functionality.

- "L" causes an "e", "f" or "g" (floating-point) conversion to assign the converted value to an object of type `long double.`

The valid conversion type specifiers are:

c Any sequence of characters in the input stream of the length specified by the field width, or a single character if no field width is specified, is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence, without a terminating null character ('\0'). For a single character assignment, a pointer to a single object of type `char` is sufficient.

C A sequence of multibyte characters in the input stream is matched. Each multibyte character is converted to a wide character of type `wchar_t.` The number of wide characters matched is specified by the field width (1 if no field width is specified). The argument is assumed to point to the first element of an array of `wchar_t` of sufficient size to contain the sequence. No terminating null wide character (L'\0') is added. For a single wide character assignment, a pointer to a single object of type `wchar_t` is sufficient.

d A decimal integer, consisting of an optional sign, followed by one or more decimal digits, is matched. The argument is assumed to point to an object of type `int.`

e, f, g A floating-point number, consisting of an optional sign ("+" or "-"), followed by one or more decimal digits, optionally containing a decimal-point character, followed by an optional exponent of the form "e" or "E", an optional sign and one or more decimal digits, is matched. The exponent, if present, specifies the power of ten by which the decimal fraction is multiplied. The argument is assumed to point to an object of type `float.`

i An optional sign, followed by an octal, decimal or hexadecimal constant is matched. An octal constant consists of "0" and zero or more octal digits. A decimal constant consists of a non-zero decimal digit and zero or more decimal digits. A hexadecimal constant consists of the characters "0x" or "0X" followed by one or more (upper- or lowercase) hexadecimal digits. The argument is assumed to point to an object of type `int.`

n No input data is processed. Instead, the number of characters that have already been read is assigned to the object of type `unsigned int` that is pointed to by the argument. The number of items that have been scanned and assigned (the return value) is not affected by the "n" conversion type specifier.

o An octal integer, consisting of an optional sign, followed by one or more (zero or non-zero) octal digits, is matched. The argument is assumed to point to an object of type `int.`

p A hexadecimal integer, as described for "x" conversions below, is matched. The converted value is further converted to a value of type `void*` and then assigned to the object pointed to by the argument.

s A sequence of non-white-space characters is matched. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

S A sequence of multibyte characters is matched. None of the multibyte characters in the sequence may be single byte white-space characters. Each multibyte character is converted to a wide character. The argument is assumed to point to the first element of an array of `wchar_t` of sufficient size to contain the sequence and a terminating null wide character, which is added by the conversion operation.

u An unsigned decimal integer, consisting of one or more decimal digits, is matched. The argument is assumed to point to an object of type `unsigned int.`

x A hexadecimal integer, consisting of an optional sign, followed by an optional prefix "0x" or "0X", followed by one or more (upper- or lowercase) hexadecimal digits, is matched. The argument is assumed to point to an object of type `int.`

[c1c2...] The longest, non-empty sequence of characters, consisting of any of the characters `c1, c2, ...` called the *scanset,* in any order, is matched. `c1` cannot be the caret character ('^'). If `c1` is "]", that character is considered to be part of the scanset and a second "]" is required to end the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

[^c1c2...] The longest, non-empty sequence of characters, consisting of any characters *other than* the characters between the "^" and "]", is matched. As with the preceding conversion, if `c1` is "]", it is considered to be part of the scanset and a second "]" ends the format directive. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating null character, which is added by the conversion operation.

For example, the specification `%[^\n]` will match an entire input line up to but not including the newline character.

A conversion type specifier of "%" is treated as a single ordinary character that matches a single "%" character in the input data. A conversion type specifier other than those listed above causes scanning to terminate the function to return.

The line

```
scanf( "%s%*f%3hx%d", name, &hexnum, &decnum )
```

with input

```
some_string 34.555e-3 abc1234
```

will copy `"some_string"` into the array `name,` skip `34.555e-3`, assign `0xabc` to `hexnum` and `1234` to `decnum.` The return value will be 3.

The program

```
#include <stdio.h>

void main()
  {
    char string1[80], string2[80];

    scanf( "%[abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWZ ]%*2s%[^\n]",
           string1, string2 );
    printf( "%s\n%s\n", string1, string2 );
  }
```

with input

```
They may look alike, but they don’t perform alike.
```

will assign

```
"They may look alike"
```

to `string1,` skip the comma (the `"%*2s"` will match only the comma; the following blank terminates that field), and assign

```
" but they don’t perform alike."
```

to `string2.`

Classification: scanf is ANSI, wscanf is ANSI
The F and N modifiers are extensions to ANSI.

Systems:
```
scanf - All, Netware
wscanf - All
```

Synopsis:

```
#include <graph.h>
void _FAR _scrolltextwindow( short rows );
```

Description: The `_scrolltextwindow` function scrolls the lines in the current text window. A text window is defined with the `_settextwindow` function. By default, the text window is the entire screen.

The argument *rows* specifies the number of rows to scroll. A positive value means to scroll the text window up or towards the top of the screen. A negative value means to scroll the text window down or towards the bottom of the screen. Specifying a number of rows greater than the height of the text window is equivalent to clearing the text window with the `_clearscreen` function.

Two constants are defined that can be used with the `_scrolltextwindow` function:

_GSCROLLUP	the contents of the text window are scrolled up (towards the top of the screen) by one row
_GSCROLLDOWN	the contents of the text window are scrolled down (towards the bottom of the screen) by one row

Returns: The `_scrolltextwindow` function does not return a value.

See Also: `_settextwindow`, `_clearscreen`, `_outtext`, `_outmem`, `_settextposition`

Example:

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

main()
{
    int i;
    char buf[ 80 ];

    _setvideomode( _TEXTC80 );
    _settextwindow( 5, 20, 20, 40 );
    for( i = 1; i <= 10; ++i ) {
        sprintf( buf, "Line %d\n", i );
        _outtext( buf );
    }
    getch();
    _scrolltextwindow( _GSCROLLDOWN );
    getch();
    _scrolltextwindow( _GSCROLLUP );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: _scrolltextwindow is PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <stdlib.h>
void _searchenv( const char *name,
                 const char *env_var,
                       char *pathname );
void _wsearchenv( const wchar_t *name,
                  const wchar_t *env_var,
                        wchar_t *pathname );
```

Description: The `_searchenv` function searches for the file specified by *name* in the list of directories assigned to the environment variable specified by *env_var*. Common values for *env_var* are PATH, LIB and INCLUDE.

The current directory is searched first to find the specified file. If the file is not found in the current directory, each of the directories specified by the environment variable is searched.

The full pathname is placed in the buffer pointed to by the argument *pathname*. If the specified file cannot be found, then *pathname* will contain an empty string.

The `_wsearchenv` function is a wide-character version of `_searchenv` that operates with wide-character strings.

Returns: The `_searchenv` function returns no value.

See Also: `getenv`, `setenv`, `_splitpath`, `putenv`

Example:

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

void display_help( FILE *fp )
  {
    printf( "display_help T.B.I.\n" );
  }
```

```
void main()
  {
    FILE *help_file;
    char full_path[ _MAX_PATH ];

    _searchenv( "watcomc.hlp", "PATH", full_path );
    if( full_path[0] == '\0' ) {
      printf( "Unable to find help file\n" );
    } else {
      help_file = fopen( full_path, "r" );
      display_help( help_file );
      fclose( help_file );
    }
  }
```

Classification: WATCOM

Systems: `_searchenv - All`
`_wsearchenv - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32`

Synopsis:
```
#include <i86.h>
void segread( struct SREGS *seg_regs );
```

Description: The `segread` function places the values of the segment registers into the structure located by *seg_regs*.

Returns: No value is returned.

See Also: `FP_OFF`, `FP_SEG`, `MK_FP`

Example:
```
#include <stdio.h>
#include <i86.h>

void main()
  {
    struct SREGS sregs;

    segread( &sregs );
    printf( "Current value of CS is %04X\n", sregs.cs );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <graph.h>
short _FAR _selectpalette( short palnum );
```

Description: The `_selectpalette` function selects the palette indicated by the argument *palnum* from the color palettes available. This function is only supported by the video modes `_MRES4COLOR` and `_MRESNOCOLOR`.

Mode `_MRES4COLOR` supports four palettes of four colors. In each palette, color 0, the background color, can be any of the 16 possible colors. The color values associated with the other three pixel values, (1, 2 and 3), are determined by the selected palette.

The following table outlines the available color palettes:

Palette	Pixel Values		
Number	1	2	3
0	green	red	brown
1	cyan	magenta	white
2	light green	light red	yellow
3	light cyan	light magenta	bright white

Returns: The `_selectpalette` function returns the number of the previously selected palette.

See Also: `_setvideomode`, `_getvideoconfig`

Example:

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

main()
{
    int x, y, pal;

    _setvideomode( _MRES4COLOR );
    for( y = 0; y < 2; ++y ) {
        for( x = 0; x < 2; ++x ) {
            _setcolor( x + 2 * y );
            _rectangle( _GFILLINTERIOR,
                    x * 160, y * 100,
                    ( x + 1 ) * 160, ( y + 1 ) * 100 );
        }
    }
    for( pal = 0; pal < 4; ++pal ) {
        _selectpalette( pal );
        getch();
    }
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
short _FAR _setactivepage( short pagenum );
```

Description: The `_setactivepage` function selects the page (in memory) to which graphics output is written. The page to be selected is given by the *pagenum* argument.

Only some combinations of video modes and hardware allow multiple pages of graphics to exist. When multiple pages are supported, the active page may differ from the visual page. The graphics information in the visual page determines what is displayed upon the screen. Animation may be accomplished by alternating the visual page. A graphics page can be constructed without affecting the screen by setting the active page to be different than the visual page.

The number of available video pages can be determined by using the `_getvideoconfig` function. The default video page is 0.

Returns: The `_setactivepage` function returns the number of the previous page when the active page is set successfully; otherwise, a negative number is returned.

See Also: `_getactivepage`, `_setvisualpage`, `_getvisualpage`, `_getvideoconfig`

Example:

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

main()
{
    int old_apage;
    int old_vpage;

    _setvideomode( _HRES16COLOR );
    old_apage = _getactivepage();
    old_vpage = _getvisualpage();
    /* draw an ellipse on page 0 */
    _setactivepage( 0 );
    _setvisualpage( 0 );
    _ellipse( _GFILLINTERIOR, 100, 50, 540, 150 );
    /* draw a rectangle on page 1 */
    _setactivepage( 1 );
    _rectangle( _GFILLINTERIOR, 100, 50, 540, 150 );
    getch();
    /* display page 1 */
    _setvisualpage( 1 );
    getch();
    _setactivepage( old_apage );
    _setvisualpage( old_vpage );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
long _FAR _setbkcolor( long color );
```

Description: The `_setbkcolor` function sets the current background color to be that of the *color* argument. In text modes, the background color controls the area behind each individual character. In graphics modes, the background refers to the entire screen. The default background color is 0.

When the current video mode is a graphics mode, any pixels with a zero pixel value will change to the color of the *color* argument. When the current video mode is a text mode, nothing will immediately change; only subsequent output is affected.

Returns: The `_setbkcolor` function returns the previous background color.

See Also: `_getbkcolor`

Example:

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

long colors[ 16 ] = {
   _BLACK, _BLUE, _GREEN, _CYAN,
   _RED, _MAGENTA, _BROWN, _WHITE,
   _GRAY, _LIGHTBLUE, _LIGHTGREEN, _LIGHTCYAN,
   _LIGHTRED, _LIGHTMAGENTA, _YELLOW, _BRIGHTWHITE
};

main()
{
    long old_bk;
    int bk;

    _setvideomode( _VRES16COLOR );
    old_bk = _getbkcolor();
    for( bk = 0; bk < 16; ++bk ) {
        _setbkcolor( colors[ bk ] );
        getch();
    }
    _setbkcolor( old_bk );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <stdio.h>
void setbuf( FILE *fp, char *buffer );
```

Description: The `setbuf` function can be used to associate a buffer with the file designated by *fp*. If this function is used, it must be called after the file has been opened and before it has been read or written. If the argument *buffer* is `NULL,` then all input/output for the file *fp* will be completely unbuffered. If the argument *buffer* is not `NULL,` then it must point to an array that is at least `BUFSIZ` characters in length, and all input/output will be fully buffered.

Returns: The `setbuf` function returns no value.

See Also: `fopen`, `setvbuf`

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

void main()
  {
    char *buffer;
    FILE *fp;

    fp = fopen( "file", "r" );
    buffer = (char *) malloc( BUFSIZ );
    setbuf( fp, buffer );
    /* . */
    /* . */
    /* . */
    fclose( fp );
  }
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <graph.h>
void _FAR _setcharsize( short height, short width );

void _FAR _setcharsize_w( double height, double width );
```

Description: The `_setcharsize` functions set the character height and width to the values specified by the arguments *height* and *width.* For the `_setcharsize` function, the arguments *height* and *width* represent a number of pixels. For the `_setcharsize_w` function, the arguments *height* and *width* represent lengths along the y-axis and x-axis in the window coordinate system.

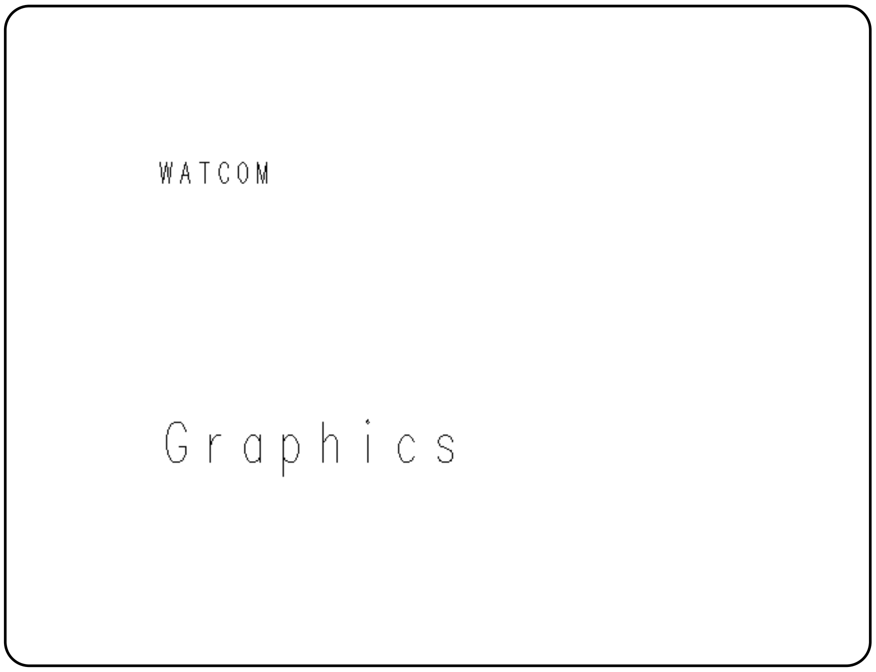
These sizes are used when displaying text with the `_grtext` function. The default character sizes are dependent on the graphics mode selected, and can be determined by the `_gettextsettings` function.

Returns: The `_setcharsize` functions do not return a value.

See Also: `_grtext`, `_gettextsettings`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    struct textsettings ts;

    _setvideomode( _VRES16COLOR );
    _gettextsettings( &ts );
    _grtext( 100, 100, "WATCOM" );
    _setcharsize( 2 * ts.height, 2 * ts.width );
    _grtext( 100, 300, "Graphics" );
    _setcharsize( ts.height, ts.width );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems:
```
_setcharsize - DOS, QNX
_setcharsize_w - DOS, QNX
```

Synopsis:

```
#include <graph.h>
void _FAR _setcharspacing( short space );

void _FAR _setcharspacing_w( double space );
```

Description: The `_setcharspacing` functions set the current character spacing to have the value of the argument *space*. For the `_setcharspacing` function, *space* represents a number of pixels. For the `_setcharspacing_w` function, *space* represents a length along the x-axis in the window coordinate system.

The character spacing specifies the additional space to leave between characters when a text string is displayed with the `_grtext` function. A negative value can be specified to cause the characters to be drawn closer together. The default value of the character spacing is 0.

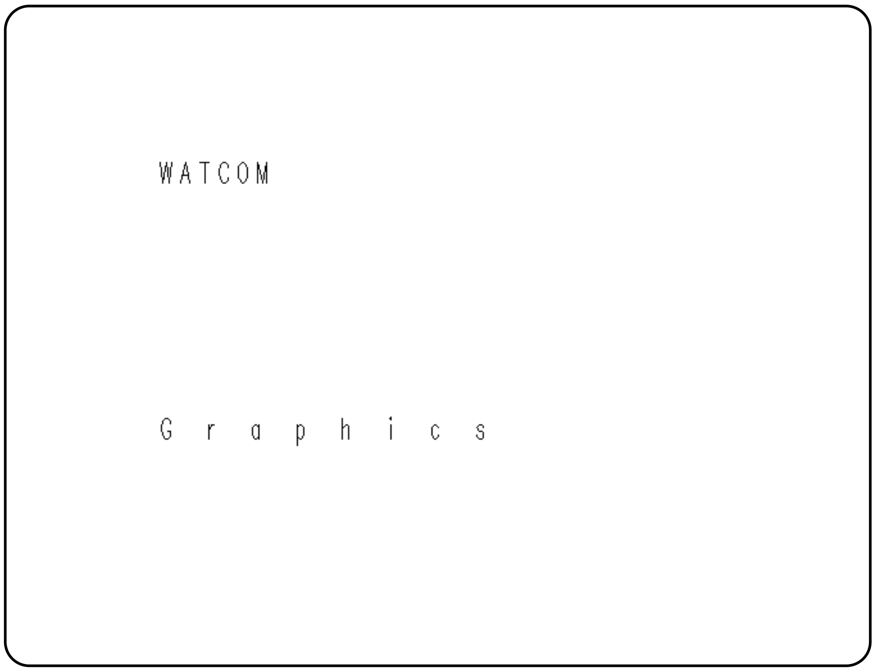
Returns: The `_setcharspacing` functions do not return a value.

See Also: `_grtext`, `_gettextsettings`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _grtext( 100, 100, "WATCOM" );
    _setcharspacing( 20 );
    _grtext( 100, 300, "Graphics" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems:
```
_setcharspacing - DOS, QNX
_setcharspacing_w - DOS, QNX
```

Synopsis:
```
#include <graph.h>
void _FAR _setcliprgn( short x1, short y1,
                       short x2, short y2 );
```

Description: The `_setcliprgn` function restricts the display of graphics output to the clipping region. This region is a rectangle whose opposite corners are established by the physical points `(x1,y1)` and `(x2,y2)`.

The `_setcliprgn` function does not affect text output using the `_outtext` and `_outmem` functions. To control the location of text output, see the `_settextwindow` function.

Returns: The `_setcliprgn` function does not return a value.

See Also: `_settextwindow`, `_setvieworg`, `_setviewport`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    short x1, y1, x2, y2;

    _setvideomode( _VRES16COLOR );
    _getcliprgn( &x1, &y1, &x2, &y2 );
    _setcliprgn( 130, 100, 510, 380 );
    _ellipse( _GBORDER, 120, 90, 520, 390 );
    getch();
    _setcliprgn( x1, y1, x2, y2 );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
short _FAR _setcolor( short pixval );
```

Description: The `_setcolor` function sets the pixel value for the current color to be that indicated by the *pixval* argument. The current color is only used by the functions that produce graphics output; text output with `_outtext` uses the current text color (see the `_settextcolor` function). The default color value is one less than the maximum number of colors in the current video mode.

Returns: The `_setcolor` function returns the previous value of the current color.

See Also: `_getcolor`, `_settextcolor`

Example:

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

main()
{
    int col, old_col;

    _setvideomode( _VRES16COLOR );
    old_col = _getcolor();
    for( col = 0; col < 16; ++col ) {
        _setcolor( col );
        _rectangle( _GFILLINTERIOR, 100, 100, 540, 380 );
        getch();
    }
    _setcolor( old_col );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <env.h>
int setenv( const char *name,
            const char *newvalue,
            int overwrite );
int _setenv( const char *name,
             const char *newvalue,
             int overwrite );
int _wsetenv( const wchar_t *name,
              const wchar_t *newvalue,
              int overwrite );
```

Description: The environment list consists of a number of environment names, each of which has a value associated with it. Entries can be added to the environment list with the DOS `set` command or with the `setenv` function. All entries in the environment list can be displayed by using the DOS `set` command with no arguments. A program can obtain the value for an environment variable by using the `getenv` function.

The `setenv` function searches the environment list for an entry of the form *name=value.* If no such string is present, `setenv` adds an entry of the form *name=newvalue* to the environment list. Otherwise, if the *overwrite* argument is non-zero, `setenv` either will change the existing value to *newvalue* or will delete the string *name=value* and add the string *name=newvalue.*

If the *newvalue* pointer is NULL, all strings of the form *name=value* in the environment list will be deleted.

The value of the pointer `environ` may change across a call to the `setenv` function.

The `setenv` function will make copies of the strings associated with *name* and *newvalue.*

The matching is case-insensitive; all lowercase letters are treated as if they were in upper case.

Entries can also be added to the environment list with the DOS `set` command or with the `putenv` or `setenv` functions. All entries in the environment list can be obtained by using the `getenv` function.

To assign a string to a variable and place it in the environment list:

```
C>SET INCLUDE=C:\WATCOM\H
```

To see what variables are in the environment list, and their current assignments:

```
C>SET
COMSPEC=C:\COMMAND.COM
PATH=C:\;C:\WATCOM
INCLUDE=C:\WATCOM\H
C>
```

The `_setenv` function is identical to `setenv`. Use `_setenv` for ANSI naming conventions.

The `_wsetenv` function is a wide-character version of `setenv` that operates with wide-character strings.

Returns: The `setenv` function returns zero upon successful completion. Otherwise, it will return a non-zero value and set `errno` to indicate the error.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

ENOMEM	Not enough memory to allocate a new environment string.

See Also: `clearenv`, `exec` Functions, `getenv`, `putenv`, `_searchenv`, `spawn` Functions, `system`

Example: The following will change the string assigned to `INCLUDE` and then display the new string.

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

void main()
  {
    char *path;

    if( setenv( "INCLUDE", "D:\\WATCOM\\H", 1 ) == 0 )
      if( (path = getenv( "INCLUDE" )) != NULL )
        printf( "INCLUDE=%s\n", path );
  }
```

Classification: WATCOM

Systems:
```
setenv - All
_setenv - All
_wsetenv - All
```

Synopsis:
```
#include <graph.h>
void _FAR _setfillmask( char _FAR *mask );
```

Description: The `_setfillmask` function sets the current fill mask to the value of the argument *mask*. When the value of the *mask* argument is `NULL,` there will be no fill mask set.

The fill mask is an eight-byte array which is interpreted as a square pattern (8 by 8) of 64 bits. Each bit in the mask corresponds to a pixel. When a region is filled, each point in the region is mapped onto the fill mask. When a bit from the mask is one, the pixel value of the corresponding point is set using the current plotting action with the current color; when the bit is zero, the pixel value of that point is not affected.

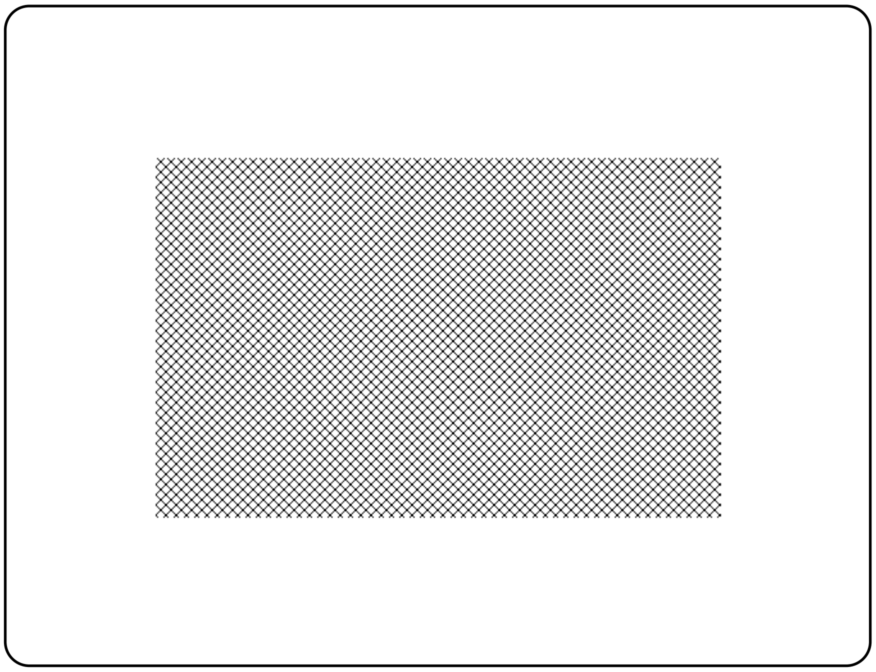
When the fill mask is not set, a fill operation will set all points in the fill region to have a pixel value of the current color. By default, no fill mask is set.

Returns: The `_setfillmask` function does not return a value.

See Also: `_getfillmask`, `_ellipse`, `_floodfill`, `_rectangle`, `_polygon`, `_pie`, `_setcolor`, `_setplotaction`

Example:
```
#include <conio.h>
#include <graph.h>

char old_mask[ 8 ];
char new_mask[ 8 ] = { 0x81, 0x42, 0x24, 0x18,
                       0x18, 0x24, 0x42, 0x81 };

main()
{
    _setvideomode( _VRES16COLOR );
    _getfillmask( old_mask );
    _setfillmask( new_mask );
    _rectangle( _GFILLINTERIOR, 100, 100, 540, 380 );
    _setfillmask( old_mask );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: _setfillmask is PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
short _FAR _setfont( char _FAR *opt );
```

Description: The `_setfont` function selects a font from the list of registered fonts (see the `_registerfonts` function). The font selected becomes the current font and is used whenever text is displayed with the `_outgtext` function. The function will fail if no fonts have been registered, or if a font cannot be found that matches the given characteristics.

The argument *opt* is a string of characters specifying the characteristics of the desired font. These characteristics determine which font is selected. The options may be separated by blanks and are not case-sensitive. Any number of options may be specified and in any order. The available options are:

hX	character height X (in pixels)
wX	character width X (in pixels)
f	choose a fixed-width font
p	choose a proportional-width font
r	choose a raster (bit-mapped) font
v	choose a vector font
b	choose the font that best matches the options
nX	choose font number X (the number of fonts is returned by the `_registerfonts` function)
t'facename'	choose a font with specified facename

The facename option is specified as a "t" followed by a facename enclosed in single quotes. The available facenames are:

Courier	fixed-width raster font with serifs
Helv	proportional-width raster font without serifs
Tms Rmn	proportional-width raster font with serifs
Script	proportional-width vector font that appears similar to hand-writing

Modern	proportional-width vector font without serifs
Roman	proportional-width vector font with serifs

When "nX" is specified to select a particular font, the other options are ignored.

If the best fit option ("b") is specified, `_setfont` will always be able to select a font. The font chosen will be the one that best matches the options specified. The following precedence is given to the options when selecting a font:

1. Pixel height (higher precedence is given to heights less than the specified height)

2. Facename

3. Pixel width

4. Font type (fixed or proportional)

When a pixel height or width does not match exactly and a vector font has been selected, the font will be stretched appropriately to match the given size.

Returns: The `_setfont` function returns zero if successful; otherwise, (-1) is returned.

See Also: `_registerfonts`, `_unregisterfonts`, `_getfontinfo`, `_outgtext`, `_getgtextextent`, `_setgtextvector`, `_getgtextvector`

Example:

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

main()
{
    int i, n;
    char buf[ 10 ];

    _setvideomode( _VRES16COLOR );
    n = _registerfonts( "*.fon" );
    for( i = 0; i < n; ++i ) {
        sprintf( buf, "n%d", i );
        _setfont( buf );
        _moveto( 100, 100 );
        _outgtext( "WATCOM Graphics" );
        getch();
        _clearscreen( _GCLEARSCREEN );
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
struct xycoord _FAR _setgtextvector( short x, short y );
```

Description: The `_setgtextvector` function sets the orientation for text output used by the `_outgtext` function to the vector specified by the arguments `(x,y).` Each of the arguments can have a value of -1, 0 or 1, allowing for text to be displayed at any multiple of a 45-degree angle. The default text orientation, for normal left-to-right text, is the vector `(1,0).`

Returns: The `_setgtextvector` function returns, as an `xycoord` structure, the previous value of the text orientation vector.

See Also: `_registerfonts`, `_unregisterfonts`, `_setfont`, `_getfontinfo`, `_outgtext`, `_getgtextextent`, `_getgtextvector`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    struct xycoord old_vec;

    _setvideomode( _VRES16COLOR );
    old_vec = _getgtextvector();
    _setgtextvector( 0, -1 );
    _moveto( 100, 100 );
    _outgtext( "WATCOM Graphics" );
    _setgtextvector( old_vec.xcoord, old_vec.ycoord );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

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

Description: The `setjmp` function saves its calling environment in its `jmp_buf` argument, for subsequent use by the `longjmp` function.

In some cases, error handling can be implemented by using `setjmp` to record the point to which a return will occur following an error. When an error is detected in a called function, that function uses `longjmp` to jump back to the recorded position. The original function which called `setjmp` must still be active (it cannot have returned to the function which called it).

Special care must be exercised to ensure that any side effects that are left undone (allocated memory, opened files, etc.) are satisfactorily handled.

Returns: The `setjmp` function returns zero when it is initially called. The return value will be non-zero if the return is the result of a call to the `longjmp` function. An `if` statement is often used to handle these two returns. When the return value is zero, the initial call to `setjmp` has been made; when the return value is non-zero, a return from a `longjmp` has just occurred.

See Also: `longjmp`

Example:
```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

rtn()
  {
    printf( "about to longjmp\n" );
    longjmp( env, 14 );
  }
```

```
void main()
  {
    int ret_val = 293;

    if( 0 == ( ret_val = setjmp( env ) ) ) {
      printf( "after setjmp %d\n", ret_val );
      rtn();
      printf( "back from rtn %d\n", ret_val );
    } else {
      printf( "back from longjmp %d\n", ret_val );
    }
  }
```

produces the following:

```
after setjmp 0
about to longjmp
back from longjmp 14
```

Classification: ANSI

Systems: MACRO

Synopsis:
```
#include <graph.h>
void _FAR _setlinestyle( unsigned short style );
```

Description: The `_setlinestyle` function sets the current line-style mask to the value of the *style* argument.

The line-style mask determines the style by which lines and arcs are drawn. The mask is treated as an array of 16 bits. As a line is drawn, a pixel at a time, the bits in this array are cyclically tested. When a bit in the array is 1, the pixel value for the current point is set using the current color according to the current plotting action; otherwise, the pixel value for the point is left unchanged. A solid line would result from a value of `0xFFFF` and a dashed line would result from a value of `0xF0F0`

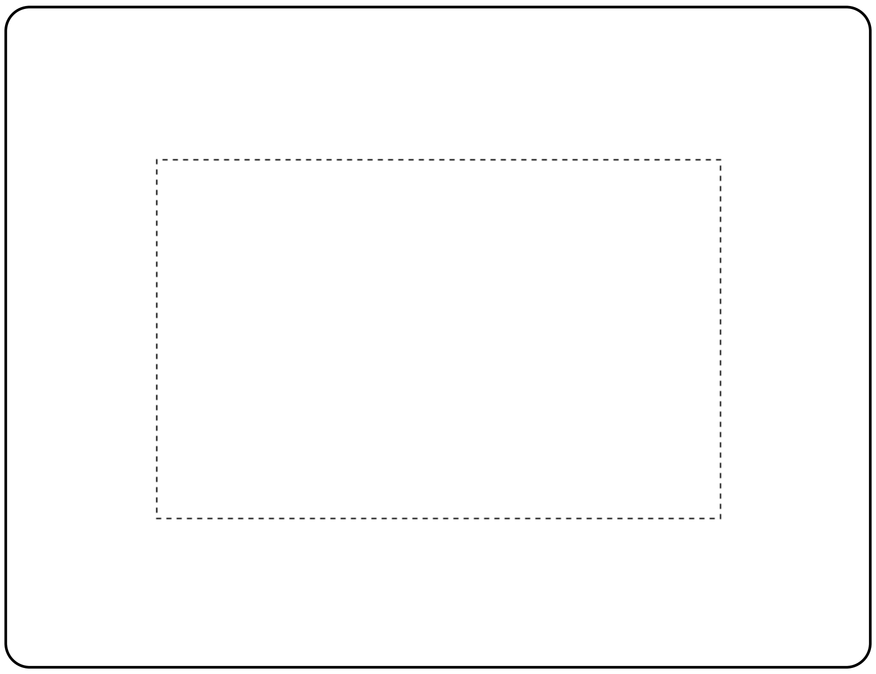
The default line style mask is `0xFFFF`

Returns: The `_setlinestyle` function does not return a value.

See Also: `_getlinestyle`, `_lineto`, `_rectangle`, `_polygon`, `_setplotaction`

Example:
```
#include <conio.h>
#include <graph.h>

#define DASHED 0xf0f0

main()
{
    unsigned old_style;

    _setvideomode( _VRES16COLOR );
    old_style = _getlinestyle();
    _setlinestyle( DASHED );
    _rectangle( _GBORDER, 100, 100, 540, 380 );
    _setlinestyle( old_style );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

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

Description: The `setlocale` function selects a portion of a program's *locale* according to the category given by *category* and the locale specified by *locale.* A *locale* affects the collating sequence (the order in which characters compare with one another), the way in which certain character-handling functions operate, the decimal-point character that is used in formatted input/output and string conversion, and the format and names used in the time string produced by the `strftime` function.

Potentially, there may be many such environments. Watcom C/C++ supports only the `"C"` locale and so invoking this function will have no effect upon the behavior of a program at present.

The possible values for the argument *category* are as follows:

Category	***Meaning***
LC_ALL	select entire environment
LC_COLLATE	select collating sequence
LC_CTYPE	select the character-handling
LC_MONETARY	select monetary formatting information
LC_NUMERIC	select the numeric-format environment
LC_TIME	select the time-related environment

At the start of a program, the equivalent of the following statement is executed.

```
setlocale( LC_ALL, "C" );
```

The `_wsetlocale` function is a wide-character version of `setlocale` that operates with wide-character strings.

Returns: If the selection is successful, a string is returned to indicate the locale that was in effect before the function was invoked; otherwise, a `NULL` pointer is returned.

See Also: `strcoll`, `strftime`, `strxfrm`

Example:

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

char src[] = { "A sample STRING" };
char dst[20];

void main()
  {
    char *prev_locale;
    size_t len;

    /* set native locale */
    prev_locale = setlocale( LC_ALL, "" );
    printf( "%s\n", prev_locale );
    len = strxfrm( dst, src, 20 );
    printf( "%s (%u)\n", dst, len );
  }
```

produces the following:

```
C
A sample STRING (15)
```

Classification: setlocale is ANSI, POSIX 1003.1, _wsetlocale is not ANSI

Systems:

```
setlocale - All, Netware
_wsetlocale - All
```

Synopsis:

```
#include <math.h>
void _set_matherr( int (*rtn)( struct exception *err_info ) )
```

Description: The default `matherr` function supplied in the library can be replaced so that the application can handle mathematical errors. To do this, the `_set_matherr` function must be called with the address of the new mathematical error handling routine.

Note: Under some systems, the default math error handler can be replaced by providing a user-written function of the same name, `matherr,` and using linking strategies to replace the default handler. Under PenPoint, the default handler is bound into a dynamic link library and can only be replaced by notifying the C library with a call to the `_set_matherr` function.

A program may contain a user-written version of `matherr` to take any appropriate action when an error is detected. When zero is returned by the user-written routine, an error message will be printed upon `stderr` and `errno` will be set as was the case with the default function. When a non-zero value is returned, no message is printed and `errno` is not changed. The value `err_info->retval` is used as the return value for the function in which the error was detected.

When called, the user-written math error handler is passed a pointer to a structure of type `struct exception` which contains information about the error that has been detected:

```
struct exception
{ int type;      /* TYPE OF ERROR                */
  char *name;    /* NAME OF FUNCTION             */
  double arg1;   /* FIRST ARGUMENT TO FUNCTION   */
  double arg2;   /* SECOND ARGUMENT TO FUNCTION  */
  double retval; /* DEFAULT RETURN VALUE         */
};
```

The `type` field will contain one of the following values:

Value	***Meaning***
DOMAIN	A domain error has occurred, such as `sqrt(-1e0).`
SING	A singularity will result, such as `pow(0e0,-2).`
OVERFLOW	An overflow will result, such as `pow(10e0,100).`
UNDERFLOW	An underflow will result, such as `pow(10e0,-100).`

TLOSS	Total loss of significance will result, such as `exp(1000).`
PLOSS	Partial loss of significance will result, such as `sin(10e70).`

The `name` field points to a string containing the name of the function which detected the error. The fields `arg1` and `arg2` (if required) give the values which caused the error. The field `retval` contains the value which will be returned by the function. This value may be changed by a user-supplied version of the `_set_matherr` function.

Returns: The `_set_matherr` function returns no value.

Example:

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

/* Demonstrate error routine in which negative */
/* arguments to "sqrt" are treated as positive */

static int my_matherr( struct exception *err );

void main()
  {
    _set_matherr( &my_matherr );
    printf( "%e\n", sqrt( -5e0 ) );
    exit( 0 );
  }

int my_matherr( struct exception *err )
  {
    if( strcmp( err->name, "sqrt" ) == 0 ) {
      if( err->type == DOMAIN ) {
        err->retval = sqrt( -(err->arg1) );
        return( 1 );
      } else
        return( 0 );
    } else
      return( 0 );
  }
```

Classification: WATCOM

Systems: Math

Synopsis:
```
#include <mbctype.h>
int _setmbcp( int codepage );
```

Description: The `_setmbcp` function sets the current code page number.

Returns: The `_setmbcp` function returns zero if the code page is set successfully. If an invalid code page value is supplied for *codepage*, the function returns -1 and the code page setting is unchanged.

See Also: `_getmbcp`, `_mbbtombc`, `_mbcjistojms`, `_mbcjmstojis`, `_mbctombb`, `_ismbbalnum`, `_ismbbalpha`, `_ismbbgraph`, `_ismbbkalnum`, `_ismbbkalpha`, `_ismbbkana`, `_ismbbkprint`, `_ismbbkpunct`, `_ismbblead`, `_ismbbprint`, `_ismbbpunct`, `_ismbbtrail`, `_mbbtombc`, `_mbcjistojms`, `_mbcjmstojis`, `_mbctombb`, `_mbbtype`

Example:
```
#include <stdio.h>
#include <mbctype.h>

void main()
  {
    printf( "%d\n", _setmbcp( 932 ) );
    printf( "%d\n", _getmbcp() );
  }
```

produces the following:

```
0
932
```

Classification: WATCOM

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <io.h>
#include <fcntl.h>
int setmode( int handle, int mode );
```

Description: The `setmode` function sets, at the operating system level, the translation mode to be the value of *mode* for the file whose file handle is given by *handle*. The mode, defined in the `<fcntl.h>` header file, can be one of:

Mode	***Meaning***
O_TEXT	On input, a carriage-return character that immediately precedes a linefeed character is removed from the data that is read. On output, a carriage-return character is inserted before each linefeed character.
O_BINARY	Data is read or written unchanged.

Returns: If successful, the `setmode` function returns the previous mode that was set for the file; otherwise, -1 is returned. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `chsize`, `close`, `creat`, `dup`, `dup2`, `eof`, `exec` Functions, `fdopen`, `filelength`, `fileno`, `fstat`, `_grow_handles`, `isatty`, `lseek`, `open`, `read`, `sopen`, `stat`, `tell`, `write`, `umask`

Example:

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

void main()
  {
    FILE *fp;
    long count;

    fp = fopen( "file", "rb" );
    if( fp != NULL ) {
      setmode( fileno( fp ), O_BINARY );
      count = 0L;
      while( fgetc( fp ) != EOF ) ++count;
      printf( "File contains %lu characters\n",
              count );
      fclose( fp );
    }
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <new.h>
PFV set_new_handler( PFV pNewHandler );
PFU _set_new_handler( PFU pNewHandler );
```

Description: The `set_new_handler` functions are used to transfer control to a user-defined error handler if the `new` operator fails to allocate memory. The argument *pNewHandler* is the name of a function of type `PFV` or `PFU.`

Type	***Description***
PFV	Pointer to a function that returns `void` (i.e., returns nothing) and takes an argument of type `void` (i.e., takes no argument).
PFU	Pointer to a function that returns `int` and takes an argument of type `unsigned` which is the amount of space to be allocated.

In a multi-threaded environment, handlers are maintained separately for each process and thread. Each new process lacks installed handlers. Each new thread gets a copy of its parent thread's new handlers. Thus, each process and thread is in charge of its own free-store error handling.

Returns: The `set_new_handler` functions return a pointer to the previous error handler so that the previous error handler can be reinstated at a later time.

The error handler specified as the argument to `_set_new_handler` returns zero indicating that further attempts to allocate memory should be halted or non-zero to indicate that an allocation request should be re-attempted.

See Also: `_bfreeseg`, `_bheapseg`, `calloc`, `free`, `malloc`, `realloc`

Example:
```
#include <stdio.h>
#include <new.h>

#if defined(__386__)
const size_t MemBlock = 8192;
#else
const size_t MemBlock = 2048;
#endif
```

```
/*
    Pre-allocate a memory block for demonstration
    purposes. The out-of-memory handler will return
    it to the system so that "new" can use it.
*/

long *failsafe = new long[MemBlock];

/*
    Declare a customized function to handle memory
    allocation failure.
*/

int out_of_memory_handler( unsigned size )
  {
    printf( "Allocation failed, " );
    printf( "%u bytes not available.\n", size );
    /* Release pre-allocated memory if we can */
    if( failsafe == NULL ) {
      printf( "Halting allocation.\n" );
      /* Tell new to stop allocation attempts */
      return( 0 );
    } else {
      delete failsafe;
      failsafe = NULL;
      printf( "Retrying allocation.\n" );
      /* Tell new to retry allocation attempt */
      return( 1 );
    }
  }

void main( void )
  {
    int i;

    /* Register existence of a new memory handler */
    _set_new_handler( out_of_memory_handler );
    long *pmemdump = new long[MemBlock];
    for( i=1 ; pmemdump != NULL; i++ ) {
      pmemdump = new long[MemBlock];
      if( pmemdump != NULL )
        printf( "Another block allocated %d\n", i );
    }
  }
```

Classification: WATCOM

Systems: set_new_handler - All, Netware
_set_new_handler - All, Netware

Synopsis:

```
#include <graph.h>
short _FAR _setpixel( short x, short y );

short _FAR _setpixel_w( double x, double y );
```

Description: The `_setpixel` function sets the pixel value of the point `(x,y)` using the current plotting action with the current color. The `_setpixel` function uses the view coordinate system. The `_setpixel_w` function uses the window coordinate system.

A pixel value is associated with each point. The values range from 0 to the number of colors (less one) that can be represented in the palette for the current video mode. The color displayed at the point is the color in the palette corresponding to the pixel number. For example, a pixel value of 3 causes the fourth color in the palette to be displayed at the point in question.

Returns: The `_setpixel` functions return the previous value of the indicated pixel if the pixel value can be set; otherwise, (-1) is returned.

See Also: `_getpixel`, `_setcolor`, `_setplotaction`

Example:

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

main()
{
    int x, y;
    unsigned i;

    _setvideomode( _VRES16COLOR );
    _rectangle( _GBORDER, 100, 100, 540, 380 );
    for( i = 0; i <= 60000; ++i ) {
        x = 101 + rand() % 439;
        y = 101 + rand() % 279;
        _setcolor( _getpixel( x, y ) + 1 );
        _setpixel( x, y );
    }
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: _setpixel is PC Graphics

Systems: `_setpixel - DOS, QNX`

```
_setpixel_w - DOS, QNX
```

Synopsis:
```
#include <graph.h>
short _FAR _setplotaction( short action );
```

Description: The `_setplotaction` function sets the current plotting action to the value of the *action* argument.

The drawing functions cause pixels to be set with a pixel value. By default, the value to be set is obtained by replacing the original pixel value with the supplied pixel value. Alternatively, the replaced value may be computed as a function of the original and the supplied pixel values.

The plotting action can have one of the following values:

_GPSET	replace the original screen pixel value with the supplied pixel value
_GAND	replace the original screen pixel value with the *bitwise and* of the original pixel value and the supplied pixel value
_GOR	replace the original screen pixel value with the *bitwise or* of the original pixel value and the supplied pixel value
_GXOR	replace the original screen pixel value with the *bitwise exclusive-or* of the original pixel value and the supplied pixel value. Performing this operation twice will restore the original screen contents, providing an efficient method to produce animated effects.

Returns: The previous value of the plotting action is returned.

See Also: `_getplotaction`

Example:

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

main()
{
    int old_act;

    _setvideomode( _VRES16COLOR );
    old_act = _getplotaction();
    _setplotaction( _GPSET );
    _rectangle( _GFILLINTERIOR, 100, 100, 540, 380 );
    getch();
    _setplotaction( _GXOR );
    _rectangle( _GFILLINTERIOR, 100, 100, 540, 380 );
    getch();
    _setplotaction( old_act );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
void _FAR _settextalign( short horiz, short vert );
```

Description: The `_settextalign` function sets the current text alignment to the values specified by the arguments *horiz* and *vert*. When text is displayed with the `_grtext` function, it is aligned (justified) horizontally and vertically about the given point according to the current text alignment settings.

The horizontal component of the alignment can have one of the following values:

_NORMAL	use the default horizontal alignment for the current setting of the text path
_LEFT	the text string is left justified at the given point
_CENTER	the text string is centred horizontally about the given point
_RIGHT	the text string is right justified at the given point

The vertical component of the alignment can have one of the following values:

_NORMAL	use the default vertical alignment for the current setting of the text path
_TOP	the top of the text string is aligned at the given point
_CAP	the cap line of the text string is aligned at the given point
_HALF	the text string is centred vertically about the given point
_BASE	the base line of the text string is aligned at the given point
_BOTTOM	the bottom of the text string is aligned at the given point

The default is to use `_LEFT` alignment for the horizontal component unless the text path is `_PATH_LEFT`, in which case `_RIGHT` alignment is used. The default value for the vertical component is `_TOP` unless the text path is `_PATH_UP`, in which case `_BOTTOM` alignment is used.

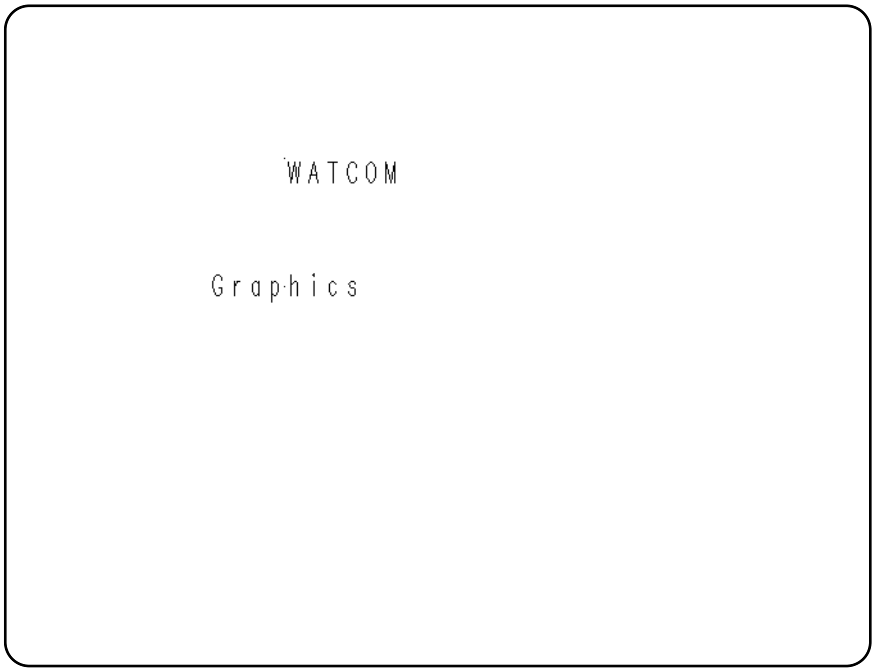
Returns: The `_settextalign` function does not return a value.

See Also: `_grtext`, `_gettextsettings`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _grtext( 200, 100, "WATCOM" );
    _setpixel( 200, 100 );
    _settextalign( _CENTER, _HALF );
    _grtext( 200, 200, "Graphics" );
    _setpixel( 200, 200 );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _settextcolor( short pixval );
```

Description: The `_settextcolor` function sets the current text color to be the color indicated by the pixel value of the *pixval* argument. This is the color value used for displaying text with the `_outtext` and `_outmem` functions. Use the `_setcolor` function to change the color of graphics output. The default text color value is set to 7 whenever a new video mode is selected.

The pixel value *pixval* is a number in the range 0-31. Colors in the range 0-15 are displayed normally. In text modes, blinking colors are specified by adding 16 to the normal color values. The following table specifies the default colors in color text modes.

Pixel value	Color	Pixel value	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	Bright White

Returns: The `_settextcolor` function returns the pixel value of the previous text color.

See Also: `_gettextcolor`, `_outtext`, `_outmem`, `_setcolor`

Example:

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

main()
{
    int old_col;
    long old_bk;

    _setvideomode( _TEXTC80 );
    old_col = _gettextcolor();
    old_bk = _getbkcolor();
    _settextcolor( 7 );
    _setbkcolor( _BLUE );
    _outtext( " WATCOM \nGraphics" );
    _settextcolor( old_col );
    _setbkcolor( old_bk );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _settextcursor( short cursor );
```

Description: The `_settextcursor` function sets the attribute, or shape, of the cursor in text modes. The argument *cursor* specifies the new cursor shape. The cursor shape is selected by specifying the top and bottom rows in the character matrix. The high byte of *cursor* specifies the top row of the cursor; the low byte specifies the bottom row.

Some typical values for *cursor* are:

Cursor	Shape
0x0607	normal underline cursor
0x0007	full block cursor
0x0407	half-height block cursor
0x2000	no cursor

Returns: The `_settextcursor` function returns the previous cursor shape when the shape is set successfully; otherwise, (-1) is returned.

See Also: `_gettextcursor`, `_displaycursor`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    int old_shape;

    old_shape = _gettextcursor();
    _settextcursor( 0x0007 );
    _outtext( "\nBlock cursor" );
    getch();
    _settextcursor( 0x0407 );
    _outtext( "\nHalf height cursor" );
    getch();
    _settextcursor( 0x2000 );
    _outtext( "\nNo cursor" );
    getch();
    _settextcursor( old_shape );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
void _FAR _settextorient( short vecx, short vecy );
```

Description: The `_settextorient` function sets the current text orientation to the vector specified by the arguments `(vecx,vecy).` The text orientation specifies the direction of the base-line vector when a text string is displayed with the `_grtext` function. The default text orientation, for normal left-to-right text, is the vector `(1,0).`

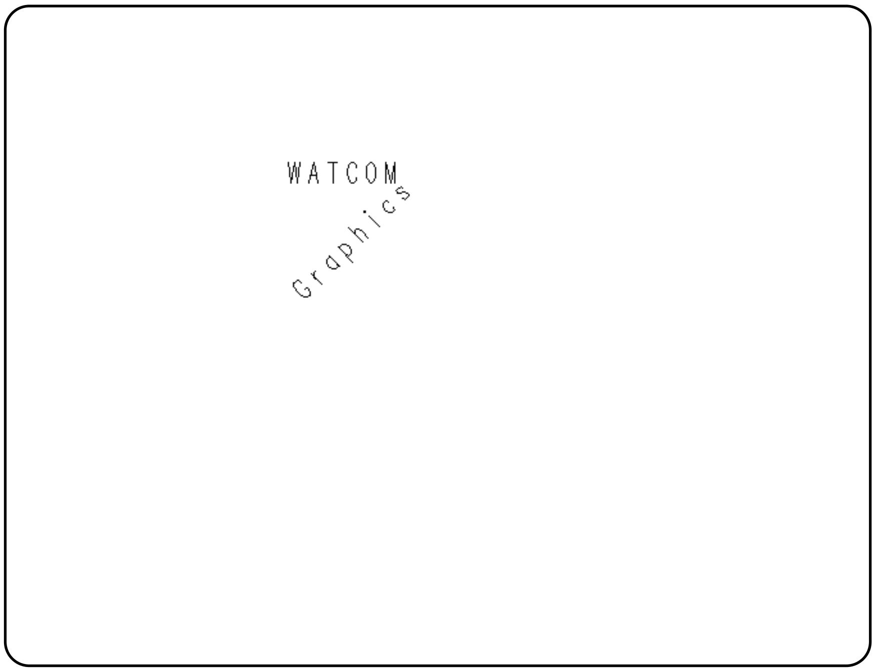
Returns: The `_settextorient` function does not return a value.

See Also: `_grtext`, `_gettextsettings`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _grtext( 200, 100, "WATCOM" );
    _settextorient( 1, 1 );
    _grtext( 200, 200, "Graphics" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
void _FAR _settextpath( short path );
```

Description: The `_settextpath` function sets the current text path to have the value of the *path* argument. The text path specifies the writing direction of the text displayed by the `_grtext` function. The argument can have one of the following values:

_PATH_RIGHT	subsequent characters are drawn to the right of the previous character
_PATH_LEFT	subsequent characters are drawn to the left of the previous character
_PATH_UP	subsequent characters are drawn above the previous character
_PATH_DOWN	subsequent characters are drawn below the previous character

The default value of the text path is `_PATH_RIGHT.`

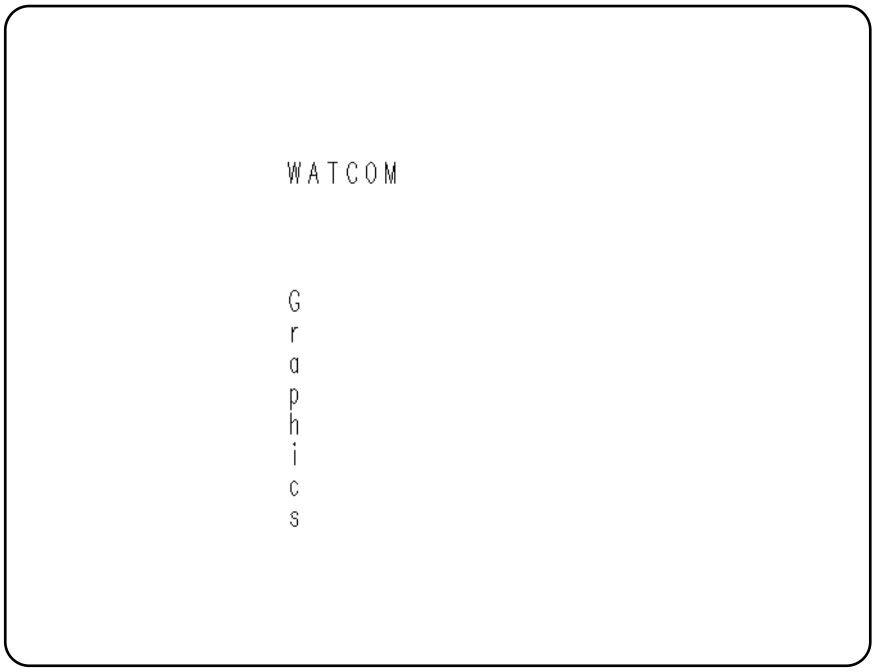
Returns: The `_settextpath` function does not return a value.

See Also: `_grtext`, `_gettextsettings`

Example:

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

main()
{
    _setvideomode( _VRES16COLOR );
    _grtext( 200, 100, "WATCOM" );
    _settextpath( _PATH_DOWN );
    _grtext( 200, 200, "Graphics" );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

produces the following:



Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
struct rccoord _FAR _settextposition( short row,
                                      short col );
```

Description: The `_settextposition` function sets the current output position for text to be `(row,col)` where this position is in terms of characters, not pixels.

The text position is relative to the current text window. It defaults to the top left corner of the screen, `(1,1),` when a new video mode is selected, or when a new text window is set. The position is updated as text is drawn with the `_outtext` and `_outmem` functions.

Note that the output position for graphics output differs from that for text output. The output position for graphics output can be set by use of the `_moveto` function.

Also note that output to the standard output file, `stdout,` is line buffered by default. It may be necessary to flush the output stream using `fflush( stdout )` after a `printf` call if your output does not contain a newline character. Mixing of calls to `_outtext` and `printf` may cause overlapped text since `_outtext` uses the output position that was set by `_settextposition`.

Returns: The `_settextposition` function returns, as an `rccoord` structure, the previous output position for text.

See Also: `_gettextposition`, `_outtext`, `_outmem`, `_settextwindow`, `_moveto`

Example:

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

main()
{
    struct rccoord old_pos;

    _setvideomode( _TEXTC80 );
    old_pos = _gettextposition();
    _settextposition( 10, 40 );
    _outtext( "WATCOM Graphics" );
    _settextposition( old_pos.row, old_pos.col );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _settextrows( short rows );
```

Description: The `_settextrows` function selects the number of rows of text displayed on the screen. The number of rows is specified by the argument *rows.* Computers equipped with EGA, MCGA and VGA adapters can support different numbers of text rows. The number of rows that can be selected depends on the current video mode and the type of monitor attached.

If the argument *rows* has the value *_MAXTEXTROWS*, the maximum number of text rows will be selected for the current video mode and hardware configuration. In text modes the maximum number of rows is 43 for EGA adapters, and 50 for MCGA and VGA adapters. Some graphics modes will support 43 rows for EGA adapters and 60 rows for MCGA and VGA adapters.

Returns: The `_settextrows` function returns the number of screen rows when the number of rows is set successfully; otherwise, zero is returned.

See Also: `_getvideoconfig`, `_setvideomode`, `_setvideomoderows`

Example:

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

int valid_rows[] = {
    14, 25, 28, 30,
    34, 43, 50, 60
};

main()
{
    int i, j, rows;
    char buf[ 80 ];

    for( i = 0; i < 8; ++i ) {
        rows = valid_rows[ i ];
        if( _settextrows( rows ) == rows ) {
            for( j = 1; j <= rows; ++j ) {
                sprintf( buf, "Line %d", j );
                _settextposition( j, 1 );
                _outtext( buf );
            }
            getch();
        }
    }
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
void _FAR _settextwindow( short row1, short col1,
                          short row2, short col2 );
```

Description: The `_settextwindow` function sets the text window to be the rectangle with a top left corner at `(row1,col1)` and a bottom right corner at `(row2,col2)`. These coordinates are in terms of characters not pixels.

The initial text output position is `(1,1)`. Subsequent text positions are reported (by the `_gettextposition` function) and set (by the `_outtext`, `_outmem` and `_settextposition` functions) relative to this rectangle.

Text is displayed from the current output position for text proceeding along the current row and then downwards. When the window is full, the lines scroll upwards one line and then text is displayed on the last line of the window.

Returns: The `_settextwindow` function does not return a value.

See Also: `_gettextposition`, `_outtext`, `_outmem`, `_settextposition`

Example:

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

main()
{
    int i;
    short r1, c1, r2, c2;
    char buf[ 80 ];

    _setvideomode( _TEXTC80 );
    _gettextwindow( &r1, &c1, &r2, &c2 );
    _settextwindow( 5, 20, 20, 40 );
    for( i = 1; i <= 20; ++i ) {
        sprintf( buf, "Line %d\n", i );
        _outtext( buf );
    }
    getch();
    _settextwindow( r1, c1, r2, c2 );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <stdio.h>
int setvbuf( FILE *fp,
             char *buf,
             int mode,
             size_t size );
```

Description: The `setvbuf` function can be used to associate a buffer with the file designated by *fp*. If this function is used, it must be called after the file has been opened and before it has been read or written. The argument *mode* determines how the file *fp* will be buffered, as follows:

Mode	***Meaning***
_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered (the buffer will be flushed when a new-line character is written, when the buffer is full, or when input is requested.
_IONBF	causes input/output to be completely unbuffered.

If the argument *buf* is not `NULL,` the array to which it points will be used instead of an automatically allocated buffer. The argument *size* specifies the size of the array.

Returns: The `setvbuf` function returns zero on success, or a non-zero value if an invalid value is given for *mode* or *size.*

See Also: `fopen`, `setbuf`

Example:

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

void main()
{
  char *buf;
  FILE *fp;

  fp = fopen( "file", "r" );
  buf = (char *) malloc( 1024 );
  setvbuf( fp, buf, _IOFBF, 1024 );
}
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <graph.h>
short _FAR _setvideomode( short mode );
```

Description: The `_setvideomode` function sets the video mode according to the value of the *mode* argument. The value of *mode* can be one of the following:

Mode	Type	Size	Colors	Adapter
_MAXRESMODE	(graphics mode with highest resolution)			
_MAXCOLORMODE	(graphics mode with most colors)			
_DEFAULTMODE	(restores screen to original mode)			
_TEXTBW40	M,T	40 x 25	16	MDPA,HGC,VGA,SVGA
_TEXTC40	C,T	40 x 25	16	CGA,EGA,MCGA,VGA,SVGA
_TEXTBW80	M,T	80 x 25	16	MDPA,HGC,VGA,SVGA
_TEXTC80	C,T	80 x 25	16	CGA,EGA,MCGA,VGA,SVGA
_MRES4COLOR	C,G	320 x 200	4	CGA,EGA,MCGA,VGA,SVGA
_MRESNOCOLOR	C,G	320 x 200	4	CGA,EGA,MCGA,VGA,SVGA
_HRESBW	C,G	640 x 200	2	CGA,EGA,MCGA,VGA,SVGA
_TEXTMONO	M,T	80 x 25	16	MDPA,HGC,VGA,SVGA
_HERCMONO	M,G	720 x 350	2	HGC
_MRES16COLOR	C,G	320 x 200	16	EGA,VGA,SVGA
_HRES16COLOR	C,G	640 x 200	16	EGA,VGA,SVGA
_ERESNOCOLOR	M,G	640 x 350	4	EGA,VGA,SVGA
_ERESCOLOR	C,G	640 x 350	4/16	EGA,VGA,SVGA
_VRES2COLOR	C,G	640 x 480	2	MCGA,VGA,SVGA
_VRES16COLOR	C,G	640 x 480	16	VGA,SVGA
_MRES256COLOR	C,G	320 x 200	256	MCGA,VGA,SVGA
_URES256COLOR	C,G	640 x 400	256	SVGA
_VRES256COLOR	C,G	640 x 480	256	SVGA
_SVRES16COLOR	C,G	800 x 600	16	SVGA
_SVRES256COLOR	C,G	800 x 600	256	SVGA
_XRES16COLOR	C,G	1024 x 768	16	SVGA
_XRES256COLOR	C,G	1024 x 768	256	SVGA

In the preceding table, the Type column contains the following letters:

M	indicates monochrome; multiple colors are shades of grey
C	indicates color
G	indicates graphics mode; size is in pixels
T	indicates text mode; size is in columns and rows of characters

The Adapter column contains the following codes:

MDPA	IBM Monochrome Display/Printer Adapter
CGA	IBM Color Graphics Adapter
EGA	IBM Enhanced Graphics Adapter
VGA	IBM Video Graphics Array
MCGA	IBM Multi-Color Graphics Array
HGC	Hercules Graphics Adapter
SVGA	SuperVGA adapters

The modes `_MAXRESMODE` and `_MAXCOLORMODE` will select from among the video modes supported by the current graphics adapter the one that has the highest resolution or the greatest number of colors. The video mode will be selected from the standard modes, not including the SuperVGA modes.

Selecting a new video mode resets the current output positions for graphics and text to be the top left corner of the screen. The background color is reset to black and the default color value is set to be one less than the number of colors in the selected mode.

Returns: The `_setvideomode` function returns the number of text rows when the new mode is successfully selected; otherwise, zero is returned.

See Also: `_getvideoconfig`, `_settextrows`, `_setvideomoderows`

Example:

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

main()
{
    int mode;
    struct videoconfig vc;
    char buf[ 80 ];

    _getvideoconfig( &vc );
    /* select "best" video mode */
    switch( vc.adapter ) {
    case _VGA :
    case _SVGA :
        mode = _VRES16COLOR;
        break;
    case _MCGA :
        mode = _MRES256COLOR;
        break;
    case _EGA :
        if( vc.monitor == _MONO ) {
            mode = _ERESNOCOLOR;
        } else {
            mode = _ERESCOLOR;
        }
        break;
    case _CGA :
        mode = _MRES4COLOR;
        break;
    case _HERCULES :
        mode = _HERCMONO;
        break;
    default :
        puts( "No graphics adapter" );
        exit( 1 );
    }
    if( _setvideomode( mode ) ) {
        _getvideoconfig( &vc );
        sprintf( buf, "%d x %d x %d\n", vc.numxpixels,
                          vc.numypixels, vc.numcolors );
        _outtext( buf );
        getch();
        _setvideomode( _DEFAULTMODE );
    }
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _setvideomoderows( short mode, short rows );
```

Description: The `_setvideomoderows` function selects a video mode and the number of rows of text displayed on the screen. The video mode is specified by the argument *mode* and is selected with the `_setvideomode` function. The number of rows is specified by the argument *rows* and is selected with the `_settextrows` function.

Computers equipped with EGA, MCGA and VGA adapters can support different numbers of text rows. The number of rows that can be selected depends on the video mode and the type of monitor attached.

Returns: The `_setvideomoderows` function returns the number of screen rows when the mode and number of rows are set successfully; otherwise, zero is returned.

See Also: `_getvideoconfig`, `_setvideomode`, `_settextrows`

Example:
```
#include <conio.h>
#include <graph.h>
#include <stdio.h>

main()
{
    int rows;
    char buf[ 80 ];

    rows = _setvideomoderows( _TEXTC80, _MAXTEXTROWS );
    if( rows != 0 ) {
        sprintf( buf, "Number of rows is %d\n", rows );
        _outtext( buf );
        getch();
        _setvideomode( _DEFAULTMODE );
    }
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
struct xycoord _FAR _setvieworg( short x, short y );
```

Description: The `_setvieworg` function sets the origin of the view coordinate system, `(0,0)`, to be located at the physical point `(x,y)`. This causes subsequently drawn images to be translated by the amount `(x,y)`.

Note: In previous versions of the software, the `_setvieworg` function was called `_setlogorg`.

Returns: The `_setvieworg` function returns, as an `xycoord` structure, the physical coordinates of the previous origin.

See Also: `_getviewcoord`, `_getphyscoord`, `_setcliprgn`, `_setviewport`

Example:
```
#include <conio.h>
#include <graph.h>

main()
{
    _setvideomode( _VRES16COLOR );
    _setvieworg( 320, 240 );
    _ellipse( _GBORDER, -200, -150, 200, 150 );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
void _FAR _setviewport( short x1, short y1,
                        short x2, short y2 );
```

Description: The _setviewport function restricts the display of graphics output to the clipping region and then sets the origin of the view coordinate system to be the top left corner of the region. This region is a rectangle whose opposite corners are established by the physical points (x1,y1) and (x2,y2).

The _setviewport function does not affect text output using the _outtext and _outmem functions. To control the location of text output, see the _settextwindow function.

Returns: The _setviewport function does not return a value.

See Also: _setcliprgn, _setvieworg, _settextwindow, _setwindow

Example:
```
#include <conio.h>
#include <graph.h>

#define XSIZE 380
#define YSIZE 280

main()
{
    _setvideomode( _VRES16COLOR );
    _setviewport( 130, 100, 130 + XSIZE, 100 + YSIZE );
    _ellipse( _GBORDER, 0, 0, XSIZE, YSIZE );
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <graph.h>
short _FAR _setvisualpage( short pagenum );
```

Description: The `_setvisualpage` function selects the page (in memory) from which graphics output is displayed. The page to be selected is given by the *pagenum* argument.

Only some combinations of video modes and hardware allow multiple pages of graphics to exist. When multiple pages are supported, the active page may differ from the visual page. The graphics information in the visual page determines what is displayed upon the screen. Animation may be accomplished by alternating the visual page. A graphics page can be constructed without affecting the screen by setting the active page to be different than the visual page.

The number of available video pages can be determined by using the `_getvideoconfig` function. The default video page is 0.

Returns: The `_setvisualpage` function returns the number of the previous page when the visual page is set successfully; otherwise, a negative number is returned.

See Also: `_getvisualpage`, `_setactivepage`, `_getactivepage`, `_getvideoconfig`

Example:

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

main()
{
    int old_apage;
    int old_vpage;

    _setvideomode( _HRES16COLOR );
    old_apage = _getactivepage();
    old_vpage = _getvisualpage();
    /* draw an ellipse on page 0 */
    _setactivepage( 0 );
    _setvisualpage( 0 );
    _ellipse( _GFILLINTERIOR, 100, 50, 540, 150 );
    /* draw a rectangle on page 1 */
    _setactivepage( 1 );
    _rectangle( _GFILLINTERIOR, 100, 50, 540, 150 );
    getch();
    /* display page 1 */
    _setvisualpage( 1 );
    getch();
    _setactivepage( old_apage );
    _setvisualpage( old_vpage );
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:
```
#include <graph.h>
short _FAR _setwindow( short invert,
                       double x1, double y1,
                       double x2, double y2 );
```

Description: The `_setwindow` function defines a window for the window coordinate system. Window coordinates are specified as a user-defined range of values. This allows for consistent pictures regardless of the video mode.

The window is defined as the region with opposite corners established by the points `(x1,y1)` and `(x2,y2).` The argument *invert* specifies the direction of the y-axis. If the value is non-zero, the y values increase from the bottom of the screen to the top, otherwise, the y values increase as you move down the screen.

The window defined by the `_setwindow` function is displayed in the current viewport. A viewport is defined by the `_setviewport` function.

By default, the window coordinate system is defined with the point `(0.0,0.0)` located at the lower left corner of the screen, and the point `(1.0,1.0)` at the upper right corner.

Returns: The `_setwindow` function returns a non-zero value when the window is set successfully; otherwise, zero is returned.

See Also: `_setviewport`

Example:

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

main()
{
    _setvideomode( _MAXRESMODE );
    draw_house( "Default window" );
    _setwindow( 1, -0.5, -0.5, 1.5, 1.5 );
    draw_house( "Larger window" );
    _setwindow( 1, 0.0, 0.0, 0.5, 1.0 );
    draw_house( "Left side" );
    _setvideomode( _DEFAULTMODE );
}

draw_house( char *msg )
{
    _clearscreen( _GCLEARSCREEN );
    _outtext( msg );
    _rectangle_w( _GBORDER, 0.2, 0.1, 0.8, 0.6 );
    _moveto_w( 0.1, 0.5 );
    _lineto_w( 0.5, 0.9 );
    _lineto_w( 0.9, 0.5 );
    _arc_w( 0.4, 0.5, 0.6, 0.3, 0.6, 0.4, 0.4, 0.4 );
    _rectangle_w( _GBORDER, 0.4, 0.1, 0.6, 0.4 );
    getch();
}
```

Classification: PC Graphics

Systems: DOS, QNX

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

Description: The `signal` function is used to specify an action to take place when certain conditions are detected while a program executes. These conditions are defined to be:

Condition	***Meaning***
SIGABRT	abnormal termination, such as caused by the `abort` function
SIGBREAK	an interactive attention (CTRL/BREAK on keyboard) is signalled
SIGFPE	an erroneous floating-point operation occurs (such as division by zero, overflow and underflow)
SIGILL	illegal instruction encountered
SIGINT	an interactive attention (CTRL/C on keyboard) is signalled
SIGSEGV	an illegal memory reference is detected
SIGTERM	a termination request is sent to the program
SIGUSR1	OS/2 process flag A via DosFlagProcess
SIGUSR2	OS/2 process flag B via DosFlagProcess
SIGUSR3	OS/2 process flag C via DosFlagProcess

An action can be specified for each of the conditions, depending upon the value of the *func* argument:

function When *func* is a function name, that function will be called equivalently to the following code sequence.

```
/* "sig_no" is condition being signalled */
signal( sig_no, SIG_DFL );
(*func)( sig_no );
```

The *func* function may terminate the program by calling the `exit` or `abort` functions or call the `longjmp` function. Because the next signal will be handled with default handling, the program must again call `signal` if it is desired to handle the next condition of the type that has been signalled.

After returning from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted.

The signal catching function is described as follows:

```
void func( int sig_no )
{

    /* body of function */

}
```

Since signal-catching functions are invoked asynchronously with process execution, the type `sig_atomic_t` may be used to define variables on which an atomic operation (e.g., incrementation, decrementation) may be performed.

SIG_DFL This value causes the default action for the condition to occur.

SIG_IGN This value causes the indicated condition to be ignored.

When a condition is detected, it may be handled by a program, it may be ignored, or it may be handled by the usual default action (often causing an error message to be printed upon the `stderr` stream followed by program termination).

When the program begins execution, the equivalent of

```
signal( SIGABRT, SIG_DFL );
signal( SIGFPE, SIG_IGN );
signal( SIGILL, SIG_DFL );
signal( SIGINT, SIG_DFL );
signal( SIGSEGV, SIG_DFL );
signal( SIGTERM, SIG_DFL );
```

is executed.

The `SIGINT` signal is generated by pressing the CTRL/C or CTRL/BREAK key combination on the keyboard. Under DOS, if "BREAK=ON", a signal will be delivered at the next DOS call; otherwise, if "BREAK=OFF", a signal will be delivered only at the next standard input/output DOS call. The `BREAK` setting is configured in the `CONFIG.SYS` file.

A condition can be generated by a program using the `raise` function.

Returns: A return value of `SIG_ERR` indicates that the request could not be handled, and `errno` is set to the value `EINVAL.`

Otherwise, the previous value of *func* for the indicated condition is returned.

See Also: `break` Functions, `raise`

Example:

```
#include <stdio.h>
#include <signal.h>
#include <i86.h>

/* SIGINT Test */

sig_atomic_t signal_count;
sig_atomic_t signal_number;

void MyIntHandler( int signo )
{
    signal_count++;
    signal_number = signo;
}

void MyBreakHandler( int signo )
{
    signal_count++;
    signal_number = signo;
}

int main( void )
{
    int i;

    signal_count = 0;
    signal_number = 0;
    signal( SIGINT, MyIntHandler );
    signal( SIGBREAK, MyBreakHandler );
    printf( "Press Ctrl/C or Ctrl/Break\n" );
    for( i = 0; i < 50; i++ ) {
        printf( "Iteration # %d\n", i );
        delay( 500 ); /* sleep for 1/2 second */
        if( signal_count > 0 ) break;
    }
    printf( "SIGINT count %d number %d\n",
                    signal_count, signal_number );
    signal_count = 0;
    signal_number = 0;
    signal( SIGINT, SIG_DFL );      /* Default action */
    signal( SIGBREAK, SIG_DFL );    /* Default action */
    printf( "Default signal handling\n" );
    for( i = 0; i < 50; i++ ) {
        printf( "Iteration # %d\n", i );
        delay( 500 ); /* sleep for 1/2 second */
        if( signal_count > 0 ) break; /* Won't happen */
    }
    return( signal_count );
}
```

Classification: ANSI

Systems: All, Netware

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

Description: The `sin` function computes the sine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns: The `sin` function returns the sine value.

See Also: `acos`, `asin`, `atan`, `atan2`, `cos`, `tan`

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

void main()
  {
    printf( "%f\n", sin(.5) );
  }
```

produces the following:

```
0.479426
```

Classification: ANSI

Systems: Math

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

Description: The `sinh` function computes the hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large.

Returns: The `sinh` function returns the hyperbolic sine value. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `ERANGE`, and print a "RANGE error" diagnostic message using the `stderr` stream.

See Also: `cosh`, `tanh`, `matherr`

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

void main()
  {
    printf( "%f\n", sinh(.5) );
  }
```

produces the following:

```
0.521095
```

Classification: ANSI

Systems: Math

Synopsis:
```
#include <wchar.h>
int sisinit( const mbstate_t *ps );
```

Description: If *ps* is not a null pointer, the `sisinit` function determines whether the pointed-to `mbstate_t` object describes an initial conversion state.

Returns: The `sisinit` function returns nonzero if *ps* is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, `wctomb`

Example:
```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>


const char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

void main()
  {
    int         i, j, k;
    wchar_t     pwc;
    mbstate_t   pstate = { 0 };

    _setmbcp( 932 );
    j = 1;
    for( i = 0; j > 0; i += j ) {
      printf( "We are %sin an initial conversion state\n",
              sisinit( &pstate ) ? "not " : "" );
      j = mbrtowc( &pwc, &chars[i], MB_CUR_MAX, &pstate );
      printf( "%d bytes in character ", j );
      if( errno == EILSEQ ) {
        printf( " - illegal multibyte character\n" );
      } else {
        if( j == 0 ) {
          k = 0;
        } else if ( j == 1 ) {
          k = chars[i];
        } else if( j == 2 ) {
          k = chars[i]<<8 | chars[i+1];
        }
        printf( "(%#6.4x->%#6.4x)\n", k, pwc );
      }
    }
  }
```

produces the following:

```
We are in an initial conversion state
1 bytes in character (0x0020->0x0020)
We are in an initial conversion state
1 bytes in character (0x002e->0x002e)
We are in an initial conversion state
1 bytes in character (0x0031->0x0031)
We are in an initial conversion state
1 bytes in character (0x0041->0x0041)
We are in an initial conversion state
2 bytes in character (0x8140->0x3000)
We are in an initial conversion state
2 bytes in character (0x8260->0xff21)
We are in an initial conversion state
2 bytes in character (0x82a6->0x3048)
We are in an initial conversion state
2 bytes in character (0x8342->0x30a3)
We are in an initial conversion state
1 bytes in character (0x00a1->0xff61)
We are in an initial conversion state
1 bytes in character (0x00a6->0xff66)
We are in an initial conversion state
1 bytes in character (0x00df->0xff9f)
We are in an initial conversion state
2 bytes in character (0xe0a1->0x720d)
We are in an initial conversion state
0 bytes in character (  0000->  0000)
```

Classification: ANSI

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <dos.h>
void sleep( unsigned seconds );
```

Description: The `sleep` function suspends execution by the specified number of *seconds.*

Returns: The `sleep` function has no return value.

See Also: `delay`

Example:
```
/*
 * The following program sleeps for the
 * number of seconds specified in argv[1].
 */
#include <stdlib.h>
#include <dos.h>

void main( int argc, char *argv[] )
  {
    unsigned seconds;

    seconds = (unsigned) strtol( argv[1], NULL, 0 );
    sleep( seconds );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <stdio.h>
int _snprintf( char *buf,
               size_t count,
               const char *format, ... );
#include <wchar.h>
int _snwprintf( wchar_t *buf,
                size_t count,
                const wchar_t *format, ... );
```

Description: The `_snprintf` function is equivalent to the `fprintf` function, except that the argument *buf* specifies a character array into which the generated output is placed, rather than to a file. A null character is placed at the end of the generated character string. The maximum number of characters to store, including a terminating null character, is specified by *count*. The *format* string is described under the description of the `printf` function.

The `_snwprintf` function is identical to `_snprintf` except that the argument *buf* specifies an array of wide characters into which the generated output is to be written, rather than converted to multibyte characters and written to a stream. The maximum number of wide characters to store, including a terminating null wide character, is specified by *count*. The `_snwprintf` function accepts a wide-character string argument for *format*

Returns: The `_snprintf` function returns the number of characters written into the array, not counting the terminating null character, or a negative value if *count* or more characters were requested to be generated. An error can occur while converting a value for output. The `_snwprintf` function returns the number of wide characters written into the array, not counting the terminating null wide character, or a negative value if *count* or more wide characters were requested to be generated. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `_vbprintf`, `vcprintf`, `vfprintf`, `vprintf`, `vsprintf`

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

/* Create temporary file names using a counter */

char namebuf[13];
int  TempCount = 0;
```

```
char *make_temp_name()
  {
    _snprintf( namebuf, 13, "ZZ%.6o.TMP", TempCount++ );
    return( namebuf );
  }

void main()
  {
    FILE *tf1, *tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );
  }
```

Classification: WATCOM

Systems:
```
_snprintf - All, Netware
_snwprintf - All
```

Synopsis:

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <sys\types.h>
#include <share.h>
int sopen( const char *filename,
           int access, int share, ... );
int _wsopen( const wchar_t *filename,
           int access, int share, ... );
```

Description: The `sopen` function opens a file at the operating system level for shared access. The name of the file to be opened is given by *filename.* The file will be accessed according to the access mode specified by *access.* When the file is to be created, the optional argument must be given which establishes the future access permissions for the file. Additionally, the sharing mode of the file is given by the *share* argument. The optional argument is the file permissions to be used when `O_CREAT` flag is on in the *access* mode.

The `_wsopen` function is identical to `sopen` except that it accepts a wide character string argument.

The access mode is established by a combination of the bits defined in the `<fcntl.h>` header file. The following bits may be set:

Mode	***Meaning***
O_RDONLY	permit the file to be only read.
O_WRONLY	permit the file to be only written.
O_RDWR	permit the file to be both read and written.
O_APPEND	causes each record that is written to be written at the end of the file.
O_CREAT	has no effect when the file indicated by *filename* already exists; otherwise, the file is created;
O_TRUNC	causes the file to be truncated to contain no data when the file exists; has no effect when the file does not exist.
O_BINARY	causes the file to be opened in binary mode which means that data will be transmitted to and from the file unchanged.

O_TEXT	causes the file to be opened in text mode which means that carriage-return characters are written before any linefeed character that is written and causes carriage-return characters to be removed when encountered during reads.
O_NOINHERIT	indicates that this file is not to be inherited by a child process.
O_EXCL	indicates that this file is to be opened for exclusive access. If the file exists and `O_CREAT` was also specified then the open will fail (i.e., use `O_EXCL` to ensure that the file does not already exist).

When neither `O_TEXT` nor `O_BINARY` are specified, the default value in the global variable `_fmode` is used to set the file translation mode. When the program begins execution, this variable has a value of `O_TEXT.`

`O_CREAT` must be specified when the file does not exist and it is to be written.

When the file is to be created (`O_CREAT` is specified), an additional argument must be passed which contains the file permissions to be used for the new file. The access permissions for the file or directory are specified as a combination of bits (defined in the `<sys\stat.h>` header file).

The following bits define permissions for the owner.

Permission	***Meaning***
S_IRWXU	Read, write, execute/search
S_IRUSR	Read permission
S_IWUSR	Write permission
S_IXUSR	Execute/search permission

The following bits define permissions for the group.

Permission	***Meaning***
S_IRWXG	Read, write, execute/search
S_IRGRP	Read permission
S_IWGRP	Write permission
S_IXGRP	Execute/search permission

The following bits define permissions for others.

Permission	***Meaning***
S_IRWXO	Read, write, execute/search
S_IROTH	Read permission
S_IWOTH	Write permission
S_IXOTH	Execute/search permission

The following bits define miscellaneous permissions used by other implementations.

Permission	***Meaning***
S_IREAD	is equivalent to S_IRUSR (read permission)
S_IWRITE	is equivalent to S_IWUSR (write permission)
S_IEXEC	is equivalent to S_IXUSR (execute/search permission)

All files are readable with DOS; however, it is a good idea to set `S_IREAD` when read permission is intended for the file.

The `sopen` function applies the current file permission mask to the specified permissions (see `umask`).

The shared access for the file, *share*, is established by a combination of bits defined in the `<share.h>` header file. The following values may be set:

Value	***Meaning***
SH_COMPAT	Set compatibility mode.
SH_DENYRW	Prevent read or write access to the file.
SH_DENYWR	Prevent write access of the file.
SH_DENYRD	Prevent read access to the file.
SH_DENYNO	Permit both read and write access to the file.

You should consult the technical documentation for the DOS system that you are using for more detailed information about these sharing modes.

Returns: If successful, `sopen` returns a handle for the file. When an error occurs while opening the file, -1 is returned. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EACCES	Access denied because *path* specifies a directory or a volume ID, or sharing mode denied due to a conflicting open.
EMFILE	No more handles available (too many open files)
ENOENT	Path or file not found

See Also: chsize, close, creat, dup, dup2, eof, exec Functions, fdopen, filelength, fileno, fstat, _grow_handles, isatty, lseek, open, read, setmode, stat, tell, write, umask

Example:

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

void main()
  {
    int handle;

    /* open a file for output                  */
    /* replace existing file if it exists      */

    handle = sopen( "file",
                O_WRONLY | O_CREAT | O_TRUNC,
                SH_DENYWR,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    /* read a file which is assumed to exist   */

    handle = sopen( "file", O_RDONLY, SH_DENYWR );

    /* append to the end of an existing file   */
    /* write a new file if file does not exist */

    handle = sopen( "file",
                O_WRONLY | O_CREAT | O_APPEND,
                SH_DENYWR,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
  }
```

Classification: WATCOM

Systems: sopen - All, Netware
_wsopen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:
```
#include <i86.h>
void sound( unsigned frequency );
```

Description: The `sound` function turns on the PC's speaker at the specified *frequency*. The frequency is in Hertz (cycles per second). The speaker can be turned off by calling the `nosound` function after an appropriate amount of time.

Returns: The `sound` function has no return value.

See Also: `delay`, `nosound`

Example:
```
#include <i86.h>

/*
    The numbers in this table are the timer divisors
    necessary to produce the pitch indicated in the
    lowest octave that is supported by the "sound"
    function.

    To raise the pitch by N octaves, simply divide the
    number in the table by 2**N since a pitch which is
    an octave above another has double the frequency of
    the original pitch.

    The frequency obtained by these numbers is given by
    1193180 / X where X is the number obtained in the
    table.
*/
```

```
unsigned short Notes[] = {
        19327 ,         /* C b            */
        18242 ,         /* C              */
        17218 ,         /* C #   ( D b )  */
        16252 ,         /* D              */
        15340 ,         /* D #   ( E b )  */
        14479 ,         /* E     ( F b )  */
        13666 ,         /* F     ( E # )  */
        12899 ,         /* F #   ( G b )  */
        12175 ,         /* G              */
        11492 ,         /* G #   ( A b )  */
        10847 ,         /* A              */
        10238 ,         /* A #   ( B b )  */
        9664 ,          /* B     ( C b )  */
        9121 ,          /* B #            */
        0
};

#define FACTOR  1193180
#define OCTAVE  4

void main()             /* play the scale */
  {
    int i;
    for( i = 0; Notes[i]; ++i ) {
      sound( FACTOR / (Notes[i] / (1 << OCTAVE)) );
      delay( 200 );
      nosound();
    }
  }
```

Classification: Intel

Systems: DOS, Windows, Win386, QNX

Synopsis:

```
#include <process.h>
int spawnl(   mode, path, arg0, arg1..., argn, NULL );
int spawnle(  mode, path, arg0, arg1..., argn, NULL, envp);
int spawnlp(  mode, file, arg0, arg1..., argn, NULL );
int spawnlpe( mode, file, arg0, arg1..., argn, NULL, envp);
int spawnv(   mode, path, argv );
int spawnve(  mode, path, argv, envp );
int spawnvp(  mode, file, argv );
int spawnvpe( mode, file, argv, envp );
  int         mode;             /* mode for parent      */
  const char *path;             /* file name incl. path */
  const char *file;             /* file name            */
  const char *arg0, ..., *argn; /* arguments            */
  const char *const argv[];     /* array of arguments   */
  const char *const envp[];     /* environment strings  */
int _wspawnl(   mode, path, arg0, arg1..., argn, NULL );
int _wspawnle(  mode, path, arg0, arg1..., argn, NULL, envp);
int _wspawnlp(  mode, file, arg0, arg1..., argn, NULL );
int _wspawnlpe( mode, file, arg0, arg1..., argn, NULL, envp);
int _wspawnv(   mode, path, argv );
int _wspawnve(  mode, path, argv, envp );
int _wspawnvp(  mode, file, argv );
int _wspawnvpe( mode, file, argv, envp );
  int            mode;             /* mode for parent      */
  const wchar_t *path;             /* file name incl. path */
  const wchar_t *file;             /* file name            */
  const wchar_t *arg0, ..., *argn; /* arguments            */
  const wchar_t *const argv[];     /* array of arguments   */
  const wchar_t *const envp[];     /* environment strings  */
```

Description: The **spawn** functions create and execute a new child process, named by *pgm*. The value of *mode* determines how the program is loaded and how the invoking program will behave after the invoked program is initiated:

Mode	***Meaning***
P_WAIT	The invoked program is loaded into available memory, is executed, and then the original program resumes execution. This option is supported under DOS, OS/2, Win32 and QNX.
P_NOWAIT	Causes the current program to execute concurrently with the new child process. This option is supported under OS/2, Win32 and QNX.

P_NOWAITO Causes the current program to execute concurrently with the new child process. This option is supported under OS/2, Win32 and QNX. The `wait` and `cwait` functions cannot be used to obtain the exit code.

P_OVERLAY The invoked program replaces the original program in memory and is executed. No return is made to the original program. This option is supported under DOS (16-bit only), OS/2, Win32, and QNX. This is equivalent to calling the appropriate `exec` function.

The program is located by using the following logic in sequence:

1. An attempt is made to locate the program in the current working directory if no directory specification precedes the program name; otherwise, an attempt is made in the specified directory.

2. If no file extension is given, an attempt is made to find the program name, in the directory indicated in the first point, with `.COM` concatenated to the end of the program name.

3. If no file extension is given, an attempt is made to find the program name, in the directory indicated in the first point, with `.EXE` concatenated to the end of the program name.

4. When no directory specification is given as part of the program name, the `spawnlp`, `spawnlpe`, `spawnvp`, and `spawnvpe` functions will repeat the preceding three steps for each of the directories specified by the `PATH` environment variable. The command

   ```
   path c:\myapps;d:\lib\applns
   ```

   indicates that the two directories

   ```
   c:\myapps
   d:\lib\applns
   ```

   are to be searched. The DOS `PATH` command (without any directory specification) will cause the current path definition to be displayed.

An error is detected when the program cannot be found.

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments in the **spawn** call. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. The length of this concatenated string must not exceed 128 bytes for DOS systems.

The arguments may be passed as a list of arguments (`spawnl`, `spawnle`, `spawnlp` and `spawnlpe`) or as a vector of pointers (`spawnv`, `spawnve`, `spawnvp`, and `spawnvpe`). At least one argument, *arg0* or *argv[0]*, must be passed to the child process. By convention, this first argument is a pointer to the name of the program.

If the arguments are passed as a list, there must be a `NULL` pointer to mark the end of the argument list. Similarly, if a pointer to an argument vector is passed, the argument vector must be terminated by a `NULL` pointer.

The environment for the invoked program is inherited from the parent process when you use the `spawnl`, `spawnlp`, `spawnv` and `spawnvp` functions. The `spawnle`, `spawnlpe`, `spawnve` and `spawnvpe` functions allow a different environment to be passed to the child process through the *envp* argument. The argument *envp* is a pointer to an array of character pointers, each of which points to a string defining an environment variable. The array is terminated with a `NULL` pointer. Each pointer locates a character string of the form

```
variable=value
```

that is used to define an environment variable. If the value of *envp* is `NULL`, then the child process inherits the environment of the parent process.

The environment is the collection of environment variables whose values that have been defined with the DOS `SET` command or by the successful execution of the `putenv` function. A program may read these values with the `getenv` function. The wide-character `_wspawnl, _wspawnle, _wspawnlp, _wspawnlpe, _wspawnv, _wspawnve, _wspawnvp` and `_wspawnvpe` functions are similar to their counterparts but operate on wide-character strings.

The following example invokes "myprog" as if `myprog ARG1 ARG2` had been entered as a command to DOS.

```
spawnl( P_WAIT, "myprog",
        "myprog", "ARG1", "ARG2", NULL );
```

The program will be found if one of "myprog.", "myprog.com", or "myprog.exe" is found in the current working directory.

The following example includes a new environment for "myprog".

```
    char *env_list[] = { "SOURCE=MYDATA",
                         "TARGET=OUTPUT",
                         "lines=65",
                         NULL
                        };

    spawnle( P_WAIT, "myprog",
            "myprog", "ARG1", "ARG2", NULL,
             env_list );
```

The environment for the invoked program will consist of the three environment variables SOURCE, TARGET and lines.

The following example is another variation on the first example.

```
    char *arg_list[] = { "myprog", "ARG1", "ARG2", NULL };

    spawnv( P_WAIT, "myprog", arg_list );
```

Returns: When the value of *mode* is:

Mode	***Meaning***
P_WAIT	then the return value from **spawn** is the exit status of the child process.
P_NOWAIT	then the return value from **spawn** is the process id (or process handle under Win32) of the child process. To obtain the exit code for a process spawned with P_NOWAIT, you must call the wait (under OS/2 or QNX) or cwait (under OS/2 or Win32) function specifying the process id/handle. If the child process terminated normally, then the low order byte of the returned status word will be set to 0, and the high order byte will contain the low order byte of the return code that the child process passed to the DOSEXIT function.
P_NOWAITO	then the return value from **spawn** is the process id of the child process. The exit code cannot be obtained for a process spawned with P_NOWAITO.

When an error is detected while invoking the indicated program, **spawn** returns -1 and errno is set to indicate the error.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
E2BIG	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EINVAL	The *mode* argument is invalid.
ENOENT	Path or file not found
ENOMEM	Not enough memory is available to execute the child process.

See Also: `abort`, `atexit`, `cwait`, `exec` Functions, `exit`, `_exit`, `getcmd`, `getenv`, `main`, `putenv`, `system`, `wait`

Example:

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

void main()
  {
    int   process_id, status, rc;

    process_id = spawnl( P_NOWAIT, "child.exe",
               "child", "5", NULL );
    if( process_id == -1 ) {
      printf( "spawn failed - %s\n", strerror( errno ) );
      exit( EXIT_FAILURE );
    }
    printf( "Process id = %d\n", process_id );
#if defined(__OS2__) || defined(__NT__)
    rc = cwait( &status, process_id, WAIT_CHILD );
    if( rc == -1 ) {
      printf( "wait failed - %s\n", strerror( errno ) );
    } else {
      printf( "wait succeeded - %x\n", status );
      switch( status & 0xff ) {
      case 0:
        printf( "Normal termination exit code = %d\n",
                status >> 8 );
        break;
      case 1:
        printf( "Hard-error abort\n" );
        break;
      case 2:
        printf( "Trap operation\n" );
        break;
      case 3:
        printf( "SIGTERM signal not intercepted\n" );
        break;
      default:
        printf( "Bogus return status\n" );
      }
    }
#endif
    printf( "spawn completed\n" );
  }

/*
[child.c]
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

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

    if( argc <= 1 ) exit( EXIT_FAILURE );
    delay = atoi( argv[1] );
    printf( "I am a child going to sleep "
            "for %d seconds\n", delay );
    sleep( delay );
    printf( "I am a child awakening\n" );
    exit( 123 );

  }
*/
```

Classification: WATCOM

Systems:

```
spawnl - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
spawnle - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
spawnlp - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32, Netware
spawnlpe - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
spawnv - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
spawnve - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
spawnvp - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32, Netware
spawnvpe - DOS, Win32, QNX, OS/2 1.x(all), OS/2-32
_wspawnl - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnle - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnlp - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnlpe - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnv - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnve - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnvp - DOS, Win32, OS/2 1.x(all), OS/2-32
_wspawnvpe - DOS, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <stdlib.h>
void _splitpath( const char *path,
                       char *drive,
                       char *dir,
                       char *fname,
                       char *ext );
void _wsplitpath( const wchar_t *path,
                        wchar_t *drive,
                        wchar_t *dir,
                        wchar_t *fname,
                        wchar_t *ext );
```

Description: The `_splitpath` function splits up a full pathname into four components consisting of a drive letter, directory path, file name and file name extension. The argument *path* points to a buffer containing the full pathname to be split up.

The `_wsplitpath` function is a wide-character version of `_splitpath` that operates with wide-character strings.

The maximum size required for each buffer is specified by the manifest constants `_MAX_PATH`, `_MAX_DRIVE` (or `_MAX_VOLUME` for Netware applications), `_MAX_DIR`, `_MAX_FNAME`, and `_MAX_EXT` which are defined in `<stdlib.h>.`

drive The *drive* argument points to a buffer that will be filled in with the drive letter (e.g., A, B, C, etc.) followed by a colon if a drive is specified in the full pathname (filled in by `_splitpath`).

For Netware applications, the *drive* argument points to a buffer that will be filled in with the volume identifier (e.g., \\NAME_SPACE) if a volume is specified in the full pathname (filled in by `_splitpath`).

dir The *dir* argument points to a buffer that will be filled in with the pathname including the trailing slash. Either forward slashes (/) or backslashes (\) may be used.

fname The *fname* argument points to a buffer that will be filled in with the base name of the file without any extension (suffix) if a file name is specified in the full pathname (filled in by `_splitpath`).

ext The *ext* argument points to a buffer that will be filled in with the filename extension (suffix) including the leading period if an extension is specified in the full pathname (filled in by `_splitpath`).

The arguments *drive, dir, fname* and *ext* will not be filled in if they are NULL pointers.

For each component of the full pathname that is not present, its corresponding buffer will be set to an empty string.

Returns: The `_splitpath` function returns no value.

See Also: `_fullpath`, `_makepath`, `_splitpath2`

Example:

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

void main()
  {
    char full_path[ _MAX_PATH ];
    char drive[ _MAX_DRIVE ];
    char dir[ _MAX_DIR ];
    char fname[ _MAX_FNAME ];
    char ext[ _MAX_EXT ];

    _makepath(full_path,"c","watcomc\\h\\","stdio","h");
    printf( "Full path is: %s\n\n", full_path );
    _splitpath( full_path, drive, dir, fname, ext );
    printf( "Components after _splitpath\n" );
    printf( "drive: %s\n", drive );
    printf( "dir:   %s\n", dir );
    printf( "fname: %s\n", fname );
    printf( "ext:   %s\n", ext );
  }
```

produces the following:

```
Full path is: c:watcomc\h\stdio.h

Components after _splitpath
drive: c:
dir:   watcomc\h\
fname: stdio
ext:   .h
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: WATCOM

Systems:

```
_splitpath - All, Netware
_wsplitpath - All
```

Synopsis:

```
#include <stdlib.h>
void _splitpath2( const char *inp,
                        char *outp,
                        char **drive,
                        char **dir,
                        char **fname,
                        char **ext );
void _wsplitpath2( const wchar_t *inp,
                         wchar_t *outp,
                         wchar_t **drive,
                         wchar_t **dir,
                         wchar_t **fname,
                         wchar_t **ext );
```

Description: The `_splitpath2` function splits up a full pathname into four components consisting of a drive letter, directory path, file name and file name extension.

inp	The argument *inp* points to a buffer containing the full pathname to be split up.
outp	The argument *outp* points to a buffer that will contain all the components of the path, each separated by a null character. The maximum size required for this buffer is specified by the manifest constant `_MAX_PATH2` which is defined in `<stdlib.h>`.
drive	The *drive* argument is the location that is to contain the pointer to the drive letter (e.g., A, B, C, etc.) followed by a colon if a drive is specified in the full pathname (filled in by `_splitpath2`).
dir	The *dir* argument is the location that is to contain the pointer to the directory path including the trailing slash if a directory path is specified in the full pathname (filled in by `_splitpath2`). Either forward slashes (/) or backslashes (\) may be used.
fname	The *fname* argument is the location that is to contain the pointer to the base name of the file without any extension (suffix) if a file name is specified in the full pathname (filled in by `_splitpath2`).
ext	The *ext* argument is the location that is to contain the pointer to the filename extension (suffix) including the leading period if an extension is specified in the full pathname (filled in by `_splitpath2`).

The arguments *drive*, *dir*, *fname* and *ext* will not be filled in if they are NULL pointers.

For each component of the full pathname that is not present, its corresponding pointer will be set to point at a NULL string ('\0').

This function reduces the amount of memory space required when compared to the `splitpath` function.

The `_wsplitpath2` function is a wide-character version of `_splitpath2` that operates with wide-character strings.

Returns: The `_splitpath2` function returns no value.

See Also: `_fullpath`, `_makepath`, `_splitpath`

Example:

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

void main()
  {
    char full_path[ _MAX_PATH ];
    char tmp_path[ _MAX_PATH2 ];
    char *drive;
    char *dir;
    char *fname;
    char *ext;

    _makepath(full_path,"c","watcomc\\h","stdio","h");
    printf( "Full path is: %s\n\n", full_path );
    _splitpath2( full_path, tmp_path,
                 &drive, &dir, &fname, &ext );
    printf( "Components after _splitpath2\n" );
    printf( "drive: %s\n", drive );
    printf( "dir:   %s\n", dir );
    printf( "fname: %s\n", fname );
    printf( "ext:   %s\n", ext );
  }
```

produces the following:

```
Full path is: c:watcomc\h\stdio.h

Components after _splitpath2
drive: c:
dir:   watcomc\h\
fname: stdio
ext:   .h
```

Note the use of two adjacent backslash characters (\) within character-string constants to signify a single backslash.

Classification: WATCOM

Systems:
```
_splitpath2 - All
_wsplitpath2 - All
```

Synopsis:
```
#include <stdio.h>
int sprintf( char *buf, const char *format, ... );
#include <wchar.h>
int swprintf( wchar_t *buf,
              size_t n,
              const wchar_t *format, ... );
```

Description: The `sprintf` function is equivalent to the `fprintf` function, except that the argument *buf* specifies a character array into which the generated output is placed, rather than to a file. A null character is placed at the end of the generated character string. The *format* string is described under the description of the `printf` function.

The `swprintf` function is identical to `sprintf` except that the argument *buf* specifies an array of wide characters into which the generated output is to be written, rather than converted to multibyte characters and written to a stream. The maximum number of wide characters to write, including a terminating null wide character, is specified by *n*. The `swprintf` function accepts a wide-character string argument for *format*

Returns: The `sprintf` function returns the number of characters written into the array, not counting the terminating null character. An error can occur while converting a value for output. The `swprintf` function returns the number of wide characters written into the array, not counting the terminating null wide character, or a negative value if *n* or more wide characters were requested to be generated. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `_vbprintf`, `vcprintf`, `vfprintf`, `vprintf`, `vsprintf`

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

/* Create temporary file names using a counter */

char namebuf[13];
int  TempCount = 0;

char *make_temp_name()
  {
    sprintf( namebuf, "ZZ%.6o.TMP", TempCount++ );
    return( namebuf );
  }
```

```
void main()
  {
    FILE *tf1, *tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );
  }
```

Classification: sprintf is ANSI, swprintf is ANSI

Systems:
```
sprintf - All, Netware
swprintf - All
```

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

Description: The `sqrt` function computes the non-negative square root of *x*. A domain error occurs if the argument is negative.

Returns: The `sqrt` function returns the value of the square root. When the argument is outside the permissible range, the `matherr` function is called. Unless the default `matherr` function is replaced, it will set the global variable `errno` to `EDOM`, and print a "DOMAIN error" diagnostic message using the `stderr` stream.

See Also: `exp`, `log`, `pow`, `matherr`

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

void main()
  {
    printf( "%f\n", sqrt(.5) );
  }
```

produces the following:

```
0.707107
```

Classification: ANSI

Systems: Math

Synopsis:

```
#include <stdlib.h>
void srand( unsigned int seed );
```

Description: The `srand` function uses the argument *seed* to start a new sequence of pseudo-random integers to be returned by subsequent calls to `rand.` A particular sequence of pseudo-random integers can be repeated by calling `srand` with the same *seed* value. The default sequence of pseudo-random integers is selected with a *seed* value of 1.

Returns: The `srand` function returns no value.

See Also: `rand`

Example:

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

void main()
  {
    int i;

    srand( 982 );
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }
    srand( 982 );  /* start sequence over again */
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }
  }
```

Classification: ANSI

Systems: All, Netware

Synopsis:

```
#include <stdio.h>
int sscanf( const char *in_string,
            const char *format, ... );
#include <wchar.h>
int swscanf( const wchar_t *in_string,
             const wchar_t *format, ... );
```

Description: The `sscanf` function scans input from the character string *in_string* under control of the argument *format*. Following the format string is the list of addresses of items to receive values.

The *format* string is described under the description of the `scanf` function.

The `swscanf` function is identical to `sscanf` except that it accepts a wide-character string argument for *format* and the input string *in_string* consists of wide characters.

Returns: The `sscanf` function returns `EOF` when the scanning is terminated by reaching the end of the input string. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

See Also: `cscanf`, `fscanf`, `scanf`, `vcscanf`, `vfscanf`, `vscanf`, `vsscanf`

Example:

```
#include <stdio.h>

/* Scan a date in the form "Saturday April 18 1987" */

void main()
  {
    int day, year;
    char weekday[10], month[10];

    sscanf( "Friday August 0014 1987",
            "%s %s %d  %d",
             weekday, month, &day, &year );
    printf( "%s %s %d %d\n",
             weekday, month, day, year );
  }
```

produces the following:

```
Friday August 14 1987
```

Classification: sscanf is ANSI, swscanf is ANSI

Systems: `sscanf - All, Netware`
`swscanf - All`

Synopsis:
```
#include <malloc.h>
size_t stackavail(void);
```

Description: The `stackavail` function returns the number of bytes currently available in the stack. This value is usually used to determine an appropriate amount to allocate using alloca.

Returns: The `stackavail` function returns the number of bytes currently available in the stack.

See Also: `alloca`, `calloc` Functions, `malloc` Functions

Example:
```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <fcntl.h>
#include <io.h>

long char_count( FILE *fp )
  {
     char *buffer;
     size_t bufsiz;
     long count;

     /* allocate half of stack for temp buffer */
     bufsiz = stackavail() >> 1;
     buffer = (char *) alloca( bufsiz );
     setvbuf( fp, buffer, _IOFBF, bufsiz );
     count = 0L;
     while( fgetc( fp ) != EOF ) ++count;
     fclose( fp );
     return( count );
  }

void main()
  {
    FILE *fp;

    fp = fopen( "file", "rb" );
    if( fp != NULL ) {
      setmode( fileno( fp ), O_BINARY );
      printf( "File contains %lu characters\n",
          char_count( fp ) );
      fclose( fp );
    }
  }
```

Classification: WATCOM

Systems: All, Netware

stat, _stat, _stati64, _wstat, _wstati64

Synopsis:

```
#include <sys\stat.h>
int stat( const char *path, struct stat *buf );
int _stat( const char *path, struct _stat *buf );
int _stati64( const char *path, struct _stati64 *buf );
int _wstat( const wchar_t *path, struct _wstat *buf );
int _wstati64( const wchar_t *path, struct _wstati64 *buf );
```

Description: The `stat` functions obtain information about the file or directory referenced in *path.* This information is placed in the structure located at the address indicated by *buf.*

The file `<sys\stat.h>` contains definitions for the structure `stat.`

Field	***Type/Meaning***
st_dev	(dev_t) the disk drive the file resides on
st_ino	(ino_t) this inode’s number (not used for DOS)
st_mode	(unsigned short) file mode
st_nlink	(short) number of hard links
st_uid	(unsigned long) user-id (always ’root’ for DOS)
st_gid	(short) group-id (always ’root’ for DOS)
st_rdev	(dev_t) this should be the device type but it is the same as st_dev for the time being
st_size	(off_t) total file size
st_atime	(time_t) this should be the file "last accessed" time if the file system supports it
st_mtime	(time_t) the file "last modified" time
st_ctime	(time_t) this should be the file "last status change" time if the file system supports it
	The following fields are Netware only:
st_btime	(time_t) the file "last archived" time

st_attr	(unsigned long) the file's attributes
st_archivedID	(unsigned long) the user/object ID that last archived file
st_updatedID	(unsigned long) the user/object ID that last updated file
st_inheritedRightsMask	(unsigned short) the inherited rights mask
st_originatingNameSpace	(unsigned char) the originating name space
st_name	(unsigned char array[_MAX_NAME]) the ASCIIZ filename (null-terminated string)

The structure `_wstat` differs from `stat` in the following way:

st_name	(wchar_t array[_MAX_NAME]) the wide character filename (null-terminated wide character string)

The structure `_stati64` differs from `stat` in the following way:

st_size	(__int64) total file size (as a 64-bit value)

The structure `_wstati64` differs from `stat` in the following ways:

st_size	(__int64) total file size (as a 64-bit value)
st_name	(wchar_t array[_MAX_NAME]) the wide character filename (null-terminated wide character string)

At least the following macros are defined in the `<sys\stat.h>` header file.

Macro	***Meaning***
S_ISFIFO(m)	Test for FIFO.
S_ISCHR(m)	Test for character special file.
S_ISDIR(m)	Test for directory file.
S_ISBLK(m)	Test for block special file.
S_ISREG(m)	Test for regular file.

The value *m* supplied to the macros is the value of the `st_mode` field of a `stat` structure. The macro evaluates to a non-zero value if the test is true and zero if the test is false.

The following bits are encoded within the `st_mode` field of a `stat` structure.

Mask	***Owner Permissions***
S_IRWXU	Read, write, search (if a directory), or execute (otherwise)
S_IRUSR	Read permission bit
S_IWUSR	Write permission bit
S_IXUSR	Search/execute permission bit
S_IREAD	== `S_IRUSR` (for Microsoft compatibility)
S_IWRITE	== `S_IWUSR` (for Microsoft compatibility)
S_IEXEC	== `S_IXUSR` (for Microsoft compatibility)

`S_IRWXU` is the bitwise inclusive OR of `S_IRUSR, S_IWUSR,` and `S_IXUSR.`

Mask	***Group Permissions (same as owner's on DOS, OS/2 or Windows)***
S_IRWXG	Read, write, search (if a directory), or execute (otherwise)
S_IRGRP	Read permission bit
S_IWGRP	Write permission bit
S_IXGRP	Search/execute permission bit

`S_IRWXG` is the bitwise inclusive OR of `S_IRGRP, S_IWGRP,` and `S_IXGRP.`

Mask	***Other Permissions (same as owner's on DOS, OS/2 or Windows)***
S_IRWXO	Read, write, search (if a directory), or execute (otherwise)
S_IROTH	Read permission bit
S_IWOTH	Write permission bit
S_IXOTH	Search/execute permission bit

`S_IRWXO` is the bitwise inclusive OR of `S_IROTH, S_IWOTH,` and `S_IXOTH.`

Mask	***Meaning***
S_ISUID	(Not supported by DOS, OS/2 or Windows) Set user ID on execution. The process's effective user ID shall be set to that of the owner of the file when the file is run as a program. On a regular file, this bit should be cleared on any write.
S_ISGID	(Not supported by DOS, OS/2 or Windows) Set group ID on execution. Set effective group ID on the process to the file's group when the file is

run as a program. On a regular file, this bit should be cleared on any write.

The `_stat` function is identical to `stat`. Use `_stat` for ANSI/ISO naming conventions. The `_fstati64`, `_wfstat`, and `_wfstati64` functions differ from `stat` in the type of structure that they are asked to fill in. The `_wfstat` and `_wfstati64` functions deal with wide character strings. The differences in the structures are described above.

Returns: All forms of the `stat` function return zero when the information is successfully obtained. Otherwise, -1 is returned.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

EACCES	Search permission is denied for a component of *path*.

See Also: `fstat`

Example:

```
#include <stdio.h>
#include <sys\stat.h>

void main()
  {
    struct stat buf;

    if( stat( "file", &buf ) != -1 ) {
      printf( "File size = %d\n", buf.st_size );
    }
  }
```

Classification: stat is POSIX 1003.1, _stat is not POSIX, _wstati64 is not POSIX

_stat conforms to ANSI/ISO naming conventions

Systems:

```
stat - All, Netware
_stat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_stati64 - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wstat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wstati64 - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <float.h>
unsigned int _status87( void );
```

Description: The `_status87` function returns the floating-point status word which is used to record the status of 8087/80287/80387/80486 floating-point operations.

Returns: The `_status87` function returns the floating-point status word which is used to record the status of 8087/80287/80387/80486 floating-point operations. The description of this status is found in the `<float.h>` header file.

See Also: `_clear87`, `_control87`, `_controlfp`, `_finite`, `_fpreset`

Example:

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

#define TEST_FPU(x,y) printf( "\t%s " y "\n", \
                ((fp_status & x) ? "  " : "No") )

void main()
  {
    unsigned int fp_status;

    fp_status = _status87();

    printf( "80x87 status\n" );
    TEST_FPU( SW_INVALID, "invalid operation" );
    TEST_FPU( SW_DENORMAL, "denormalized operand" );
    TEST_FPU( SW_ZERODIVIDE, "divide by zero" );
    TEST_FPU( SW_OVERFLOW, "overflow" );
    TEST_FPU( SW_UNDERFLOW, "underflow" );
    TEST_FPU( SW_INEXACT, "inexact result" );
  }
```

Classification: Intel

Systems: Math

Synopsis:

```
#include <string.h>
char *strcat( char *dst, const char *src );
char __far *_fstrcat( char __far *dst,
                      const char __far *src );
#include <wchar.h>
wchar_t *wcscat( wchar_t *dst, const wchar_t *src );
#include <mbstring.h>
unsigned char *_mbscat( unsigned char *dst,
                  const unsigned char *src );
unsigned char __far *_fmbscat( unsigned char __far *dst,
                         const unsigned char __far *src );
```

Description: The `strcat` function appends a copy of the string pointed to by *src* (including the terminating null character) to the end of the string pointed to by *dst*. The first character of *src* overwrites the null character at the end of *dst*.

The `_fstrcat` function is a data model independent form of the `strcat` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcscat` function is a wide-character version of `strcat` that operates with wide-character strings.

The `_mbscat` function is a multibyte character version of `strcat` that operates with multibyte character strings.

Returns: The value of *dst* is returned.

See Also: `strncat`

Example:

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

void main()
  {
    char buffer[80];

    strcpy( buffer, "Hello " );
    strcat( buffer, "world" );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
Hello world
```

Classification: strcat is ANSI, _fstrcat is not ANSI, wcscat is ANSI, _mbscat is not ANSI, _fmbscat is not ANSI

Systems:

```
strcat - All, Netware
_fstrcat - All
wcscat - All
_mbscat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbscat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strchr( const char *s, int c );
char __far *_fstrchr( const char __far *s, int c );
#include <wchar.h>
wchar_t *wcschr( const wchar_t *s, int c );
#include <mbstring.h>
unsigned char *_mbschr( const unsigned char *s,
                        unsigned int c );
unsigned char __far *_fmbschr(
            const unsigned char __far *s,
            unsigned int c );
```

Description: The `strchr` function locates the first occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

The `_fstrchr` function is a data model independent form of the `strchr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcschr` function is a wide-character version of `strchr` that operates with wide-character strings.

The `_mbschr` function is a multibyte character version of `strchr` that operates with multibyte character strings.

Returns: The `strchr` function returns a pointer to the located character, or `NULL` if the character does not occur in the string.

See Also: `memchr`, `strcspn`, `strrchr`, `strspn`, `strstr`, `strtok`

Example:

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

void main()
  {
    char buffer[80];
    char *where;

    strcpy( buffer, "video x-rays" );
    where = strchr( buffer, 'x' );
    if( where == NULL ) {
        printf( "'x' not found\n" );
    }
  }
```

Classification: strchr is ANSI, _fstrchr is not ANSI, wcschr is ANSI, _mbschr is not ANSI, _fmbschr is not ANSI

Systems:

```
strchr - All, Netware
_fstrchr - All
wcschr - All
_mbschr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbschr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
int strcmp( const char *s1, const char *s2 );
int _fstrcmp( const char __far *s1,
              const char __far *s2 );
#include <wchar.h>
int wcscmp( const wchar_t *s1, const wchar_t *s2 );
#include <mbstring.h>
int _mbscmp( const unsigned char *s1,
             const unsigned char *s2 );
int _fmbscmp( const unsigned char __far *s1,
              const unsigned char __far *s2 );
```

Description: The `strcmp` function compares the string pointed to by *s1* to the string pointed to by *s2*.

The `_fstrcmp` function is a data model independent form of the `strcmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `wcscmp` function is a wide-character version of `strcmp` that operates with wide-character strings.

The `_mbscmp` function is a multibyte character version of `strcmp` that operates with multibyte character strings.

Returns: The `strcmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

See Also: `strcmpi`, `stricmp`, `strncmp`, `strnicmp`

Example:

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

void main()
  {
    printf( "%d\n", strcmp( "abcdef", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "abc" ) );
    printf( "%d\n", strcmp( "abc", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "mnopqr" ) );
    printf( "%d\n", strcmp( "mnopqr", "abcdef" ) );
  }
```

produces the following:

```
0
1
-1
-1
1
```

Classification: strcmp is ANSI, _fstrcmp is not ANSI, wcscmp is ANSI, _mbscmp is not ANSI, _fmbscmp is not ANSI

Systems:

```
strcmp - All, Netware
_fstrcmp - All
wcscmp - All
_mbscmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbscmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <string.h>
int strcmpi( const char *s1, const char *s2 );
int wcscmpi( const wchar_t *s1, const wchar_t *s2 );
```

Description: The `strcmpi` function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*. All uppercase characters from *s1* and *s2* are mapped to lowercase for the purposes of doing the comparison. The `strcmpi` function is identical to the `stricmp` function.

The `wcscmpi` function is a wide-character version of `strcmpi` that operates with wide-character strings.

Returns: The `strcmpi` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

See Also: `strcmp`, `stricmp`, `strncmp`, `strnicmp`

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

void main()
  {
    printf( "%d\n", strcmpi( "AbCDEF", "abcdef" ) );
    printf( "%d\n", strcmpi( "abcdef", "ABC"    ) );
    printf( "%d\n", strcmpi( "abc",    "ABCdef" ) );
    printf( "%d\n", strcmpi( "Abcdef", "mnopqr" ) );
    printf( "%d\n", strcmpi( "Mnopqr", "abcdef" ) );
  }
```

produces the following:

```
0
100
-100
-12
12
```

Classification: WATCOM

Systems:
```
strcmpi - All, Netware
wcscmpi - All
```

Synopsis:
```
#include <string.h>
int strcoll( const char *s1, const char *s2 );
#include <wchar.h>
int wcscoll( const wchar_t *s1, const wchar_t *s2 );
#include <mbstring.h>
int _mbscoll( const unsigned char *s1, const unsigned char *s2
 );
```

Description: The `strcoll` function compares the string pointed to by *s1* to the string pointed to by *s2*. The comparison uses the collating sequence selected by the `setlocale` function. The function will be equivalent to the `strcmp` function when the collating sequence is selected from the `"C"` locale.

The `wcscoll` function is a wide-character version of `strcoll` that operates with wide-character strings.

The `_mbscoll` function is a multibyte character version of `strcoll` that operates with multibyte character strings.

Returns: The `strcoll` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, according to the collating sequence selected.

See Also: `setlocale`, `strcmp`, `strncmp`

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

char buffer[80] = "world";

void main()
  {
    if( strcoll( buffer, "Hello" ) < 0 ) {
        printf( "Less than\n" );
    }
  }
```

Classification: strcoll is ANSI, wcscoll is ANSI, _mbscoll is not ANSI

Systems:
```
strcoll - All, Netware
wcscoll - All
_mbscoll - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strcpy( char *dst, const char *src );
char __far *_fstrcpy( char __far *dst,
                      const char __far *src );
#include <wchar.h>
wchar_t *wcscpy( wchar_t *dst, const wchar_t *src );
#include <mbstring.h>
int _mbscpy( unsigned char *dst,
             const unsigned char *src );
int _fmbscpy( unsigned char __far *dst,
              const unsigned char __far *src );
```

Description: The `strcpy` function copies the string pointed to by *src* (including the terminating null character) into the array pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly. See the description for the `memmove` function to copy objects that overlap.

The `_fstrcpy` function is a data model independent form of the `strcpy` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcscpy` function is a wide-character version of `strcpy` that operates with wide-character strings.

The `_mbscpy` function is a multibyte character version of `strcpy` that operates with multibyte character strings.

Returns: The value of *dst* is returned.

See Also: `strdup`, `strncpy`

Example:

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

void main()
  {
    auto char buffer[80];

    strcpy( buffer, "Hello " );
    strcat( buffer, "world" );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
Hello world
```

Classification: strcpy is ANSI, _fstrcpy is not ANSI, wcscpy is ANSI, _mbscpy is not ANSI, _fmbscpy is not ANSI

Systems:

```
strcpy - All, Netware
_fstrcpy - All
wcscpy - All
_mbscpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbscpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <string.h>
size_t strcspn( const char *str,
                const char *charset );
size_t _fstrcspn( const char __far *str,
                  const char __far *charset );
#include <wchar.h>
size_t wcscspn( const wchar_t *str,
                const wchar_t *charset );
#include <mbstring.h>
size_t _mbscpsn( const unsigned char *str,
                 const unsigned char *charset );
size_t _fmbscpsn( const unsigned char __far *str,
                  const unsigned char __far *charset );
```

Description: The `strcspn` function computes the length, in bytes, of the initial segment of the string pointed to by *str* which consists entirely of characters *not* from the string pointed to by *charset.* The terminating null character is not considered part of *str*.

The `_fstrcspn` function is a data model independent form of the `strcspn` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `wcscspn` function is a wide-character version of `strcspn` that operates with wide-character strings.

The `_mbscspn` function is a multibyte character version of `strcspn` that operates with multibyte character strings.

Returns: The length, in bytes, of the initial segment is returned.

See Also: `strspn`

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

void main()
  {
    printf( "%d\n", strcspn( "abcbcadef", "cba" ) );
    printf( "%d\n", strcspn( "xxxbcadef", "cba" ) );
    printf( "%d\n", strcspn( "123456789", "cba" ) );
  }
```

produces the following:

```
0
3
9
```

Classification: strcspn is ANSI, _fstrcspn is not ANSI, wcscspn is ANSI, _mbscspn is not ANSI, _fmbscspn is not ANSI

Systems:
```
strcspn - All, Netware
_fstrcspn - All
wcscspn - All
_mbscspn - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbscspn - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <time.h>
char *_strdate( char *datestr )
wchar_t _wstrdate( wchar_t *datestr );
```

Description: The `_strdate` function copies the current date to the buffer pointed to by *datestr*. The date is formatted as "MM/DD/YY" where "MM" is two digits representing the month, where "DD" is two digits representing the day, and where "YY" is two digits representing the year. The buffer must be at least 9 bytes long.

The `_wstrdate` function is a wide-character version of `_strdate` that operates with wide-character strings.

Returns: The `_strdate` function returns a pointer to the resulting text string *datestr*.

See Also: `asctime`, `ctime`, `gmtime`, `localtime`, `mktime`, `_strtime`, `time`, `tzset`

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

void main()
  {
    char datebuff[9];

    printf( "%s\n", _strdate( datebuff ) );
  }
```

Classification: WATCOM

Systems:
```
_strdate - All
_wstrdate - All
```

Synopsis:

```
#include <tchar.h>
char *_strdec( const char *start, const char *current );
wchar_t *_wcsdec( const wchar_t *start,
                  const wchar_t *current );
#include <mbstring.h>
unsigned char *_mbsdec( const unsigned char *start,
                        const unsigned char *current );
unsigned char *_fmbsdec( const unsigned char __far *start,
                     const unsigned char __far *current );
```

Description: The `_strdec` function returns a pointer to the previous character (single-byte, wide, or multibyte) in the string pointed to by *start* which must precede *current.* The current character in the string is pointed to by *current.* You must ensure that *current* does not point into the middle of a multibyte or wide character.

The function is a data model independent form of the `_strdec` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsdec` function is a wide-character version of `_strdec` that operates with wide-character strings.

The `_mbsdec` function is a multibyte character version of `_strdec` that operates with multibyte character strings.

Returns: The `_strdec` function returns a pointer to the previous character (single-byte, wide, or multibyte depending on the function used).

See Also: `_strinc`, `_strninc`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int                 j, k;
    const unsigned char *prev;

    _setmbcp( 932 );
    prev = &chars[ SIZE - 1 ];
    do {
      prev = _mbsdec( chars, prev );
      j = mblen( prev, MB_CUR_MAX );
      if( j == 0 ) {
        k = 0;
      } else if ( j == 1 ) {
        k = *prev;
      } else if( j == 2 ) {
        k = *(prev)<<8 | *(prev+1);
      }
      printf( "Previous character %#6.4x\n", k );
    } while( prev != chars );
  }
```

produces the following:

```
Previous character 0xe0a1
Previous character 0x00df
Previous character 0x00a6
Previous character 0x00a1
Previous character 0x8342
Previous character 0x82a6
Previous character 0x8260
Previous character 0x8140
Previous character 0x0041
Previous character 0x0031
Previous character 0x002e
Previous character 0x0020
```

Classification: WATCOM

Systems:
```
_strdec - MACRO
_wcsdec - MACRO
_mbsdec - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsdec - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strdup( const char *src );
char *_strdup( const char *src );
char __far *_fstrdup( const char __far *src );
#include <wchar.h>
wchar_t *_wcsdup( const wchar_t *src );
#include <mbstring.h>
unsigned char *_mbsdup( unsigned char *src );
unsigned char __far *_fmbsdup( unsigned char __far *src );
```

Description: The `strdup` function creates a duplicate copy of the string pointed to by *src* and returns a pointer to the new copy. For `strdup`, the memory for the new string is obtained by using the `malloc` function and can be freed using the `free` function. For `_fstrdup`, the memory for the new string is obtained by using the `_fmalloc` function and can be freed using the `_ffree` function.

The `_strdup` function is identical to `strdup`. Use `_strdup` for ANSI/ISO naming conventions.

The `_fstrdup` function is a data model independent form of the `strdup` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsdup` function is a wide-character version of `strdup` that operates with wide-character strings.

The `_mbsdup` function is a multibyte character version of `strdup` that operates with multibyte character strings.

The `_fmbsdup` function is a data model independent form of the `_mbsdup` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strdup` function returns the pointer to the new copy of the string if successful, otherwise it returns `NULL.`

See Also: `free`, `malloc`, `strcpy`, `strncpy`

Example:

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

void main()
  {
    char *dup;
```

```
        dup = strdup( "Make a copy" );
        printf( "%s\n", dup );
      }
```

Classification: WATCOM

_strdup conforms to ANSI/ISO naming conventions

Systems:

```
strdup - All, Netware
_strdup - All, Netware
_fstrdup - All
_wcsdup - All
_mbsdup - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsdup - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <string.h>
char *strerror( int errnum );
wchar_t *wcserror( int errnum );
```

Description: The `strerror` function maps the error number contained in *errnum* to an error message.

The `wcserror` function is identical to `strerror` except that the message it points to is a wide-character string.

Returns: The `strerror` function returns a pointer to the error message. The array containing the error string should not be modified by the program. This array may be overwritten by a subsequent call to the `strerror` function.

See Also: `clearerr`, `feof`, `ferror`, `perror`

Example:
```
#include <stdio.h>
#include <string.h>
#include <errno.h>

void main()
  {
    FILE *fp;

    fp = fopen( "file.nam", "r" );
    if( fp == NULL ) {
        printf( "Unable to open file: %s\n",
                 strerror( errno ) );
    }
  }
```

Classification: strerror is ANSI, wcserror is ANSI

Systems:
```
strerror - All, Netware
wcserror - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <time.h>
size_t strftime( char *s,
                 size_t maxsize,
                 const char *format,
                 const struct tm *timeptr );
#include <wchar.h>
size_t wcsftime( wchar_t *s,
                 size_t maxsize,
                 const wchar_t *format,
                 const struct tm *timeptr );
#include <time.h>
size_t _wstrftime_ms( wchar_t *s,
                      size_t maxsize,
                      const char *format,
                      const struct tm *timeptr );

struct  tm {
  int tm_sec;   /* seconds after the minute -- [0,61] */
  int tm_min;   /* minutes after the hour   -- [0,59] */
  int tm_hour;  /* hours after midnight     -- [0,23] */
  int tm_mday;  /* day of the month         -- [1,31] */
  int tm_mon;   /* months since January     -- [0,11] */
  int tm_year;  /* years since 1900                   */
  int tm_wday;  /* days since Sunday        -- [0,6]  */
  int tm_yday;  /* days since January 1     -- [0,365]*/
  int tm_isdst; /* Daylight Savings Time flag */
};
```

Description: The `strftime` function formats the time in the argument *timeptr* into the array pointed to by the argument *s* according to the *format* argument.

The `wcsftime` function is a wide-character version of `strftime` that operates with wide-character strings.

The `_wstrftime_ms` function is identical to `wcsftime` except that the *format* is not a wide-character string.

The *format* string consists of zero or more directives and ordinary characters. A directive consists of a ’%’ character followed by a character that determines the substitution that is to take place. All ordinary characters are copied unchanged into the array. No more than *maxsize* characters are placed in the array. The format directives %D, %h, %n, %r, %t, and %T are from POSIX.

Directive	***Meaning***
%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 as a decimal number (01-31)
%D	date in the format mm/dd/yy (POSIX)
%h	locale’s abbreviated month name (POSIX)
%H	hour (24-hour clock) as a decimal number (00-23)
%I	hour (12-hour clock) as a decimal number (01-12)
%j	day of the year as a decimal number (001-366)
%m	month as a decimal number (01-12)
%M	minute as a decimal number (00-59)
%n	newline character (POSIX)
%p	locale’s equivalent of either AM or PM
%r	12-hour clock time (01-12) using the AM/PM notation in the format HH:MM:SS (AM|PM) (POSIX)
%S	second as a decimal number (00-59)
%t	tab character (POSIX)
%T	24-hour clock time in the format HH:MM:SS (POSIX)
%U	week number of the year as a decimal number (00-52) where Sunday is the first day of the week

%w	weekday as a decimal number (0-6) where 0 is Sunday
%W	week number of the year as a decimal number (00-52) where Monday is the first day of the week
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year without century as a decimal number (00-99)
%Y	year with century as a decimal number
%Z, %z	timezone name, or by no characters if no timezone exists (%z is an extension to ANSI/POSIX)
%%	character %

When the `%Z` or `%z` directive is specified, the `tzset` function is called.

Returns: If the number of characters to be placed into the array is less than *maxsize,* the `strftime` function returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `setlocale`, `asctime`, `clock`, `ctime`, `difftime`, `gmtime`, `localtime`, `mktime`, `time`, `tzset`

Example:

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

void main()
  {
    time_t time_of_day;
    char buffer[ 80 ];

    time_of_day = time( NULL );
    strftime( buffer, 80, "Today is %A %B %d, %Y",
               localtime( &time_of_day ) );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
Today is Friday December 25, 1987
```

Classification: strftime is ANSI, POSIX, wcsftime is ANSI, _wstrftime_ms is not ANSI

Systems:
```
strftime - All, Netware
wcsftime - All
_wstrftime_ms - All
```

Synopsis:

```
#include <string.h>
int stricmp( const char *s1, const char *s2 );
int _stricmp( const char *s1, const char *s2 );
int _fstricmp( const char __far *s1,
               const char __far *s2 );
#include <wchar.h>
int _wcsicmp( const wchar_t *s1, const wchar_t *s2 );
#include <mbstring.h>
int _mbsicmp( const unsigned char *s1,
              const unsigned char *s2 );
int _fmbsicmp( const unsigned char __far *s1,
               const unsigned char __far *s2 );
```

Description: The `stricmp` function compares, with case insensitivity, the string pointed to by *s1* to the string pointed to by *s2*. All uppercase characters from *s1* and *s2* are mapped to lowercase for the purposes of doing the comparison.

The `_stricmp` function is identical to `stricmp`. Use `_stricmp` for ANSI/ISO naming conventions.

The `_fstricmp` function is a data model independent form of the `stricmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsicmp` function is a wide-character version of `stricmp` that operates with wide-character strings.

The `_mbsicmp` function is a multibyte character version of `stricmp` that operates with multibyte character strings.

The `_fmbsicmp` function is a data model independent form of the `_mbsicmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `stricmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

See Also: `strcmp`, `strcmpi`, `strncmp`, `strnicmp`

Example:

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

void main()
  {
    printf( "%d\n", stricmp( "AbCDEF", "abcdef" ) );
    printf( "%d\n", stricmp( "abcdef", "ABC"    ) );
    printf( "%d\n", stricmp( "abc",    "ABCdef" ) );
    printf( "%d\n", stricmp( "Abcdef", "mnopqr" ) );
    printf( "%d\n", stricmp( "Mnopqr", "abcdef" ) );
  }
```

produces the following:

```
0
100
-100
-12
12
```

Classification: WATCOM

_stricmp conforms to ANSI/ISO naming conventions

Systems:

```
stricmp - All, Netware
_stricmp - All, Netware
_fstricmp - All
_wcsicmp - All
_mbsicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
int _stricoll( const char *s1, const char *s2 );
#include <wchar.h>
int _wcsicoll( const wchar_t *s1, const wchar_t *s2 );
#include <mbstring.h>
int _mbsicoll( const unsigned char *s1, const unsigned char *s
2 );
```

Description: The `_stricoll` function performs a case insensitive comparison of the string pointed to by *s1* to the string pointed to by *s2*. The comparison uses the current code page which can be selected by the `_setmbcp` function.

The `_wcsicoll` function is a wide-character version of `_stricoll` that operates with wide-character strings.

The `_mbsicoll` function is a multibyte character version of `_stricoll` that operates with multibyte character strings.

Returns: These functions return an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, according to the collating sequence selected.

See Also: `_setmbcp`, `strcoll`, `stricmp`, `strncmp`, `_strncoll`, `strnicmp`, `_strnicoll`

Example:

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

char buffer[80] = "world";

void main()
  {
    int test;

    test = _stricoll( buffer, "world2" );
    if( test < 0 ) {
        printf( "Less than\n" );
    } else if( test == 0 ) {
        printf( "Equal\n" );
    } else {
        printf( "Greater than\n" );
    }
  }
```

Classification: WATCOM

Systems: `_stricoll - All, Netware`
`_wcsicoll - All`
`_mbsicoll - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32`

Synopsis:

```
#include <tchar.h>
char *_strinc( const char *current );
wchar_t *_wcsinc( const wchar_t *current );
#include <mbstring.h>
unsigned char *_mbsinc( const unsigned char *current );
unsigned char *_fmbsinc(
                    const unsigned char __far *current );
```

Description: The `_strinc` function returns a pointer to the next character (single-byte, wide, or multibyte) in the string pointed to by *current*. You must ensure that *current* does not point into the middle of a multibyte or wide character.

The function is a data model independent form of the `_strinc` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsinc` function is a wide-character version of `_strinc` that operates with wide-character strings.

The `_mbsinc` function is a multibyte character version of `_strinc` that operates with multibyte character strings.

Returns: The `_strinc` function returns a pointer to the next character (single-byte, wide, or multibyte depending on the function used).

See Also: `_strdec`, `_strninc`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int                 j, k;
    const unsigned char *next;

    _setmbcp( 932 );
    next = chars;
    do {
      next = _mbsinc( next );
      j = mblen( next, MB_CUR_MAX );
      if( j == 0 ) {
        k = 0;
      } else if ( j == 1 ) {
        k = *next;
      } else if( j == 2 ) {
        k = *(next)<<8 | *(next+1);
      }
      printf( "Next character %#6.4x\n", k );
    } while( next != &chars[ SIZE - 1 ] );
  }
```

produces the following:

```
Next character 0x002e
Next character 0x0031
Next character 0x0041
Next character 0x8140
Next character 0x8260
Next character 0x82a6
Next character 0x8342
Next character 0x00a1
Next character 0x00a6
Next character 0x00df
Next character 0xe0a1
Next character   0000
```

Classification: WATCOM

Systems:
```
_strinc - MACRO
_wcsinc - MACRO
_mbsinc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsinc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
size_t strlen( const char *s );
size_t _fstrlen( const char __far *s );
#include <wchar.h>
size_t wcslen( const wchar_t *s );
#include <mbstring.h>
size_t _mbslen( const unsigned char *s );
size_t _fmbslen( const unsigned char __far *s );
```

Description: The `strlen` function computes the length of the string pointed to by *s*.

The `_fstrlen` function is a data model independent form of the `strlen` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `wcslen` function is a wide-character version of `strlen` that operates with wide-character strings.

The `_mbslen` function is a multibyte character version of `strlen` that operates with multibyte character strings.

The `_fmbslen` function is a data model independent form of the `_mbslen` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strlen` function returns the number of characters that precede the terminating null character.

Example:

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

void main()
  {
    printf( "%d\n", strlen( "Howdy" ) );
    printf( "%d\n", strlen( "Hello world\n" ) );
    printf( "%d\n", strlen( "" ) );
  }
```

produces the following:

```
5
12
0
```

Classification: strlen is ANSI, _fstrlen is not ANSI, wcslen is ANSI, _mbslen is not ANSI, _fmbslen is not ANSI

Systems:
```
strlen - All, Netware
_fstrlen - All
wcslen - All
_mbslen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbslen - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strlwr( char *s1 );
char *_strlwr( char *s1 );
char __far *_fstrlwr( char __far *s1 );
#include <wchar.h>
wchar_t *_wcslwr( wchar_t *s1 );
#include <mbstring.h>
unsigned char *_mbslwr( unsigned char *s1 );
unsigned char __far *_fmbslwr( unsigned char __far *s1 );
```

Description: The `strlwr` function replaces the string *s1* with lowercase characters by invoking the `tolower` function for each character in the string.

The `_strlwr` function is identical to `strlwr`. Use `_strlwr` for ANSI/ISO naming conventions.

The `_fstrlwr` function is a data model independent form of the `strlwr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `_wcslwr` function is a wide-character version of `strlwr` that operates with wide-character strings.

The `_mbslwr` function is a multibyte character version of `strlwr` that operates with multibyte character strings.

The `_fmbslwr` function is a data model independent form of the `_mbslwr` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The address of the original string *s1* is returned.

See Also: `strupr`

Example:

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

char source[] = { "A mixed-case STRING" };

void main()
  {
    printf( "%s\n", source );
    printf( "%s\n", strlwr( source ) );
    printf( "%s\n", source );
  }
```

produces the following:

```
A mixed-case STRING
a mixed-case string
a mixed-case string
```

Classification: WATCOM

_strlwr conforms to ANSI/ISO naming conventions

Systems:

```
strlwr - All, Netware
_strlwr - All, Netware
_fstrlwr - All
_wcslwr - All
_mbslwr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbslwr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strncat( char *dst, const char *src, size_t n );
char __far *_fstrncat( char __far *dst,
                 const char __far *src,
                       size_t n );
#include <wchar.h>
wchar_t *wcsncat( wchar_t *dst,
            const wchar_t *src,
                  size_t n );
#include <mbstring.h>
unsigned char *_mbsncat( unsigned char *dst,
                   const unsigned char *src,
                         size_t n );
unsigned char __far *_fmbsncat( unsigned char __far *dst,
                          const unsigned char __far *src,
                                size_t n );
```

Description: The `strncat` function appends not more than *n* characters of the string pointed to by *src* to the end of the string pointed to by *dst.* The first character of *src* overwrites the null character at the end of *dst.* A terminating null character is always appended to the result.

The `_fstrncat` function is a data model independent form of the `strncat` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcsncat` function is a wide-character version of `strncat` that operates with wide-character strings.

The `_mbsncat` function is a multibyte character version of `strncat` that operates with multibyte character strings.

The `_fmbsncat` function is a data model independent form of the `_mbsncat` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strncat` function returns the value of *dst.*

See Also: `strcat`

Example:

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

char buffer[80];

void main()
  {
    strcpy( buffer, "Hello " );
    strncat( buffer, "world", 8 );
    printf( "%s\n", buffer );
    strncat( buffer, "**************", 4 );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
Hello world
Hello world****
```

Classification: strncat is ANSI, _fstrncat is not ANSI, wcsncat is ANSI, _mbsncat is not ANSI, _fmbsncat is not ANSI

Systems:

```
strncat - All, Netware
_fstrncat - All
wcsncat - All
_mbsncat - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsncat - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
int strncmp( const char *s1,
             const char *s2,
             size_t n );
int _fstrncmp( const char __far *s1,
               const char __far *s2,
               size_t n );
#include <wchar.h>
int wcsncmp( const wchar_t *s1,
             const wchar_t *s2,
             size_t n );
#include <mbstring.h>
int _mbsncmp( const unsigned char *s1,
              const unsigned char *s2,
              size_t n );
int _fmbsncmp( const unsigned char __far *s1,
               const unsigned char __far *s2,
               size_t n );
```

Description: The `strncmp` compares not more than *n* characters from the string pointed to by *s1* to the string pointed to by *s2.*

The `_fstrncmp` function is a data model independent form of the `strncmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `wcsncmp` function is a wide-character version of `strncmp` that operates with wide-character strings.

The `_mbsncmp` function is a multibyte character version of `strncmp` that operates with multibyte character strings.

The `_fmbsncmp` function is a data model independent form of the `_mbsncmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strncmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2.*

See Also: `strcmp`, `stricmp`, `strnicmp`

Example:

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

void main()
  {
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 10 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF",  6 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF",  3 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF",  0 ) );
  }
```

produces the following:

```
1
1
0
0
```

Classification: strncmp is ANSI, _fstrncmp is not ANSI, wcsncmp is ANSI, _mbsncmp is not ANSI, _fmbsncmp is not ANSI

Systems:

```
strncmp - All, Netware
_fstrncmp - All
wcsncmp - All
_mbsncmp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsncmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
int _strncoll( const char *s1,
               const char *s2,
               size_t count );
#include <wchar.h>
int _wcsncoll( const wchar_t *s1,
               const wchar_t *s2,
               size_t count );
#include <mbstring.h>
int _mbsncoll( const unsigned char *s1,
               const unsigned char *s2,
               size_t count );
```

Description: These functions compare the first *count* characters of the string pointed to by *s1* to the string pointed to by *s2*. The comparison uses the current code page which can be selected by the `_setmbcp` function.

The `_wcsncoll` function is a wide-character version of `_strncoll` that operates with wide-character strings.

The `_mbsncoll` function is a multibyte character version of `_strncoll` that operates with multibyte character strings.

Returns: These functions return an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, according to the collating sequence selected.

See Also: `_setmbcp`, `strcoll`, `stricmp`, `_stricoll`, `strncmp`, `strnicmp`, `_strnicoll`

Example:

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

char buffer[80] = "world";

void main()
  {
    int test;

    test = _strncoll( buffer, "world2", 5 );
    if( test < 0 ) {
        printf( "Less than\n" );
    } else if( test == 0 ) {
        printf( "Equal\n" );
    } else {
        printf( "Greater than\n" );
    }
  }
```

Classification: WATCOM

Systems:

```
_strncoll - All, Netware
_wcsncoll - All
_mbsncoll - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
char *strncpy( char *dst,
               const char *src,
               size_t n );
char __far *_fstrncpy( char __far *dst,
                       const char __far *src,
                       size_t n );
#include <wchar.h>
wchar_t *wcsncpy( wchar_t *dst,
                  const wchar_t *src,
                  size_t n );
#include <mbstring.h>
unsigned char *_mbsncpy( unsigned char *dst,
                   const unsigned char *src,
                         size_t n );
unsigned char __far *_fmbsncpy( unsigned char __far *dst,
                          const unsigned char __far *src,
                                size_t n );
```

Description: The `strncpy` function copies no more than *n* characters from the string pointed to by *src* into the array pointed to by *dst*. Copying of overlapping objects is not guaranteed to work properly. See the `memmove` function if you wish to copy objects that overlap.

If the string pointed to by *src* is shorter than *n* characters, null characters are appended to the copy in the array pointed to by *dst*, until *n* characters in all have been written. If the string pointed to by *src* is longer than *n* characters, then the result will not be terminated by a null character.

The `_fstrncpy` function is a data model independent form of the `strncpy` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcsncpy` function is a wide-character version of `strncpy` that operates with wide-character strings.

The `_mbsncpy` function is a multibyte character version of `strncpy` that operates with multibyte character strings.

The `_fmbsncpy` function is a data model independent form of the `_mbsncpy` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strncpy` function returns the value of *dst*.

See Also: `strcpy`, `strdup`

Example:

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

void main()
  {
    char buffer[15];

    printf( "%s\n", strncpy( buffer, "abcdefg", 10 ) );
    printf( "%s\n", strncpy( buffer, "1234567",  6 ) );
    printf( "%s\n", strncpy( buffer, "abcdefg",  3 ) );
    printf( "%s\n", strncpy( buffer, "*******",  0 ) );
  }
```

produces the following:

```
abcdefg
123456g
abc456g
abc456g
```

Classification: strncpy is ANSI, _fstrncpy is not ANSI, wcsncpy is ANSI, _mbsncpy is not ANSI, _fmbsncpy is not ANSI

Systems:

```
strncpy - All, Netware
_fstrncpy - All
wcsncpy - All
_mbsncpy - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsncpy - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
int strnicmp( const char *s1,
              const char *s2,
              size_t len );
int _strnicmp( const char *s1,
               const char *s2,
               size_t len );
int _fstrnicmp( const char __far *s1,
                const char __far *s2,
                size_t len );
#include <wchar.h>
int _wcsnicmp( const wchar_t *s1,
               const wchar_t *s2,
               size_t len );
#include <mbstring.h>
int _mbsnicmp( const unsigned char *s1,
               const unsigned char *s2,
               size_t n );
int _fmbsnicmp( const unsigned char __far *s1,
                const unsigned char __far *s2,
                size_t n );
```

Description: The `strnicmp` function compares, without case sensitivity, the string pointed to by *s1* to the string pointed to by *s2*, for at most *len* characters.

The `_strnicmp` function is identical to `strnicmp`. Use `_strnicmp` for ANSI/ISO naming conventions.

The `_fstrnicmp` function is a data model independent form of the `strnicmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsnicmp` function is a wide-character version of `strnicmp` that operates with wide-character strings.

The `_mbsnicmp` function is a multibyte character version of `strnicmp` that operates with multibyte character strings.

The `_fmbsnicmp` function is a data model independent form of the `_mbsnicmp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strnicmp` function returns an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

See Also: `strcmp`, `stricmp`, `strncmp`

Example:

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

void main()
  {
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 10 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX",  6 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX",  3 ) );
    printf( "%d\n", strnicmp( "abcdef", "ABCXXX",  0 ) );
  }
```

produces the following:

```
-20
-20
0
0
```

Classification: WATCOM

_strnicmp conforms to ANSI/ISO naming conventions

Systems:

```
strnicmp - All, Netware
_strnicmp - All, Netware
_fstrnicmp - All
_wcsnicmp - All
_mbsnicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
_fmbsnicmp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
int _strnicoll( const char *s1,
                const char *s2,
                size_t count );
#include <wchar.h>
int _wcsnicoll( const wchar_t *s1,
                const wchar_t *s2,
                size_t count );
#include <mbstring.h>
int _mbsnicoll( const unsigned char *s1,
                const unsigned char *s2,
                size_t count );
```

Description: These functions perform a case insensitive comparison of the first *count* characters of the string pointed to by *s1* to the string pointed to by *s2*. The comparison uses the current code page which can be selected by the `_setmbcp` function.

The `_wcsnicoll` function is a wide-character version of `_strnicoll` that operates with wide-character strings.

The `_mbsnicoll` function is a multibyte character version of `_strnicoll` that operates with multibyte character strings.

Returns: These functions return an integer less than, equal to, or greater than zero, indicating that the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, according to the collating sequence selected.

See Also: `_setmbcp`, `strcoll`, `stricmp`, `_stricoll`, `strncmp`, `_strncoll`, `strnicmp`

Example:

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

char buffer[80] = "world";

void main()
  {
    int test;

    test = _strnicoll( buffer, "World2", 5 );
    if( test < 0 ) {
        printf( "Less than\n" );
    } else if( test == 0 ) {
        printf( "Equal\n" );
    } else {
        printf( "Greater than\n" );
    }
  }
```

Classification: WATCOM

Systems:
```
_strnicoll - All, Netware
_wcsnicoll - All
_mbsnicoll - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <tchar.h>
char *_strninc( const char *str, size_t count );
wchar_t *_wcsninc( const wchar_t *str, size_t count );
#include <mbstring.h>
unsigned char *_mbsninc( const unsigned char *str,
                         size_t count );
unsigned char __far *_fmbsninc(
                        const unsigned char __far *str,
                        size_t count );
```

Description: The `_mbsninc` function increments *str* by *count* multibyte characters. `_mbsninc` recognizes multibyte-character sequences according to the multibyte code page currently in use. The header file `<tchar.h>` defines the generic-text routine `_tcsninc.` This macro maps to `_mbsninc` if `_MBCS` has been defined, or to `_wcsninc` if `_UNICODE` has been defined. Otherwise `_tcsninc` maps to `_strninc`. `_strninc` and `_wcsninc` are single-byte-character string and wide-character string versions of `_mbsninc`. `_wcsninc` and `_strninc` are provided only for this mapping and should not be used otherwise.

Returns: The `_strninc` function returns a pointer to *str* after it has been incremented by *count* characters or `NULL` if *str* was `NULL.` If *count* exceeds the number of characters remaining in the string, the result is undefined.

See Also: `_strdec`, `_strinc`

Example:

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

const unsigned char chars[] = {
    ' ',
    '.',
    '1',
    'A',
    0x81,0x40, /* double-byte space */
    0x82,0x60, /* double-byte A */
    0x82,0xA6, /* double-byte Hiragana */
    0x83,0x42, /* double-byte Katakana */
    0xA1,      /* single-byte Katakana punctuation */
    0xA6,      /* single-byte Katakana alphabetic */
    0xDF,      /* single-byte Katakana alphabetic */
    0xE0,0xA1, /* double-byte Kanji */
    0x00
};

#define SIZE sizeof( chars ) / sizeof( unsigned char )

void main()
  {
    int                 j, k;
    const unsigned char *next;

    _setmbcp( 932 );
    next = chars;
    do {
      next = _mbsninc( next, 1 );
      j = mblen( next, MB_CUR_MAX );
      if( j == 0 ) {
        k = 0;
      } else if ( j == 1 ) {
        k = *next;
      } else if( j == 2 ) {
        k = *(next)<<8 | *(next+1);
      }
      printf( "Next character %#6.4x\n", k );
    } while( next != &chars[ SIZE - 1 ] );
  }
```

produces the following:

```
Next character 0x002e
Next character 0x0031
Next character 0x0041
Next character 0x8140
Next character 0x8260
Next character 0x82a6
Next character 0x8342
Next character 0x00a1
Next character 0x00a6
Next character 0x00df
Next character 0xe0a1
Next character   0000
```

Classification: WATCOM

Systems:

```
_strninc - MACRO
_wcsninc - MACRO
_mbsninc - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsninc - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <string.h>
char *strnset( char *str, int fill, size_t count );
char *_strnset( char *str, int fill, size_t count );
char __far *_fstrnset( char __far *str,
                       int fill,
                       size_t count );
#include <wchar.h>
wchar_t *_wcsnset( wchar_t *str, int fill, size_t count );
#include <mbstring.h>
unsigned char *_mbsnset( unsigned char *str,
                         unsigned int fill,
                         size_t count );
unsigned char __far *_fmbsnset( unsigned char __far *str,
                                unsigned int fill,
                                size_t __n );
```

Description: The `strnset` function fills the string *str* with the value of the argument *fill*, converted to be a character value. When the value of *count* is greater than the length of the string, the entire string is filled. Otherwise, that number of characters at the start of the string are set to the fill character.

The `_strnset` function is identical to `strnset`. Use `_strnset` for ANSI naming conventions.

The `_fstrnset` function is a data model independent form of the `strnset` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `_wcsnset` function is a wide-character version of `strnset` that operates with wide-character strings. For `_wcsnset`, the value of *count* is the number of wide characters to fill. This is half the number of bytes.

The `_mbsnset` function is a multibyte character version of `strnset` that operates with multibyte character strings.

The `_fmbsnset` function is a data model independent form of the `_mbsnset` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

For `_mbsnset`, the value of *count* is the number of multibyte characters to fill. If the number of bytes to be filled is odd and *fill* is a double-byte character, the partial byte at the end is filled with an ASCII space character.

Returns: The address of the original string *str* is returned.

See Also: strset

Example:

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

char source[] = { "A sample STRING" };

void main()
  {
    printf( "%s\n", source );
    printf( "%s\n", strnset( source, '=', 100 ) );
    printf( "%s\n", strnset( source, '*', 7 ) );
  }
```

produces the following:

```
A sample STRING
===============
*******========
```

Classification: WATCOM

Systems:

```
strnset - All, Netware
_strnset - All, Netware
_fstrnset - All
_wcsnset - All
_mbsnset - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsnset - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
char *strpbrk( const char *str, const char *charset );
char __far *_fstrpbrk( const char __far *str,
                       const char __far *charset );
#include <wchar.h>
wchar_t *wcspbrk( const wchar_t *str,
                  const wchar_t *charset );
#include <mbstring.h>
unsigned char *_mbspbrk( const unsigned char *str,
                         const unsigned char *charset );
unsigned char __far *_fmbspbrk(
                    const unsigned char __far *str,
                    const unsigned char __far *charset );
```

Description: The `strpbrk` function locates the first occurrence in the string pointed to by *str* of any character from the string pointed to by *charset.*

The `_fstrpbrk` function is a data model independent form of the `strpbrk` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcspbrk` function is a wide-character version of `strpbrk` that operates with wide-character strings.

The `_mbspbrk` function is a multibyte character version of `strpbrk` that operates with multibyte character strings.

The `_fmbspbrk` function is a data model independent form of the `_mbspbrk` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strpbrk` function returns a pointer to the located character, or `NULL` if no character from *charset* occurs in *str.*

See Also: `strchr`, `strrchr`, `strtok`

Example:

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

void main()
  {
    char *p = "Find all vowels";
```

```
        while( p != NULL ) {
          printf( "%s\n", p );
          p = strpbrk( p+1, "aeiouAEIOU" );
        }
      }
```

produces the following:

```
Find all vowels
ind all vowels
all vowels
owels
els
```

Classification: strpbrk is ANSI, _fstrpbrk is not ANSI, wcspbrk is ANSI, _mbspbrk is not ANSI, _fmbspbrk is not ANSI

Systems:
```
strpbrk - All, Netware
_fstrpbrk - All
wcspbrk - All
_mbspbrk - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbspbrk - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
char *strrchr( const char *s, int c );
char __far *_fstrrchr( const char __far *s, int c );
#include <wchar.h>
wchar_t *wcsrchr( const wchar_t *s, wint_t c );
#include <mbstring.h>
unsigned char *_mbsrchr( const unsigned char *s,
                         unsigned int c );
unsigned char __far *_fmbsrchr(
                        const unsigned char __far *s,
                        unsigned int c );
```

Description: The `strrchr` function locates the last occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating null character is considered to be part of the string.

The `_fstrrchr` function is a data model independent form of the `strrchr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcsrchr` function is a wide-character version of `strrchr` that operates with wide-character strings.

The `_mbsrchr` function is a multibyte character version of `strrchr` that operates with multibyte character strings.

The `_fmbsrchr` function is a data model independent form of the `_mbsrchr` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strrchr` function returns a pointer to the located character, or a NULL pointer if the character does not occur in the string.

See Also: `strchr`, `strpbrk`

Example:

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

void main()
  {
    printf( "%s\n", strrchr( "abcdeabcde", ’a’ ) );
    if( strrchr( "abcdeabcde", ’x’ ) == NULL )
        printf( "NULL\n" );
  }
```

produces the following:

```
abcde
NULL
```

Classification: strrchr is ANSI, _fstrrchr is not ANSI, wcsrchr is ANSI, _mbsrchr is not ANSI, _fmbsrchr is not ANSI

Systems:
```
strrchr - All, Netware
_fstrrchr - All
wcsrchr - All
_mbsrchr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsrchr - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <string.h>
char *strrev( char *s1 );
char *_strrev( char *s1 );
char __far *_fstrrev( char __far *s1 );
#include <wchar.h>
wchar_t *_wcsrev( wchar_t *s1 );
#include <mbstring.h>
unsigned char *_mbsrev( unsigned char *s1 );
unsigned char __far *_fmbsrev( unsigned char __far *s1 );
```

Description: The `strrev` function replaces the string *s1* with a string whose characters are in the reverse order.

The `_strrev` function is identical to `strrev`. Use `_strrev` for ANSI/ISO naming conventions.

The `_fstrrev` function is a data model independent form of the `strrev` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `_wcsrev` function is a wide-character version of `strrev` that operates with wide-character strings.

The `_mbsrev` function is a multibyte character version of `strrev` that operates with multibyte character strings.

The `_fmbsrev` function is a data model independent form of the `_mbsrev` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The address of the original string *s1* is returned.

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

char source[] = { "A sample STRING" };

void main()
  {
    printf( "%s\n", source );
    printf( "%s\n", strrev( source ) );
    printf( "%s\n", strrev( source ) );
  }
```

produces the following:

```
A sample STRING
GNIRTS elpmas A
A sample STRING
```

Classification: WATCOM

_strrev conforms to ANSI/ISO naming conventions

Systems:
```
strrev - All, Netware
_strrev - All, Netware
_fstrrev - All
_wcsrev - All
_mbsrev - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsrev - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strset( char *s1, int fill );
char *_strset( char *s1, int fill );
char __far *_fstrset( char __far *s1, int fill );
#include <wchar.h>
wchar_t *_wcsset( wchar_t *s1, int fill );
#include <mbstring.h>
unsigned char *_mbsset( unsigned char *s1,
                        unsigned int fill );
unsigned char __far *_fmbsset( unsigned char __far *s1,
                               unsigned int fill );
```

Description: The `strset` function fills the string pointed to by *s1* with the character *fill*. The terminating null character in the original string remains unchanged.

The `_strset` function is identical to `strset`. Use `_strset` for ANSI naming conventions.

The `_fstrset` function is a data model independent form of the `strset` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `_wcsset` function is a wide-character version of `strset` that operates with wide-character strings.

The `_mbsset` function is a multibyte character version of `strset` that operates with multibyte character strings.

The `_fmbsset` function is a data model independent form of the `_mbsset` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The address of the original string *s1* is returned.

See Also: `strnset`

Example:

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

char source[] = { "A sample STRING" };

void main()
  {
    printf( "%s\n", source );
    printf( "%s\n", strset( source, '=' ) );
    printf( "%s\n", strset( source, '*' ) );
  }
```

produces the following:

```
A sample STRING
===============
***************
```

Classification: WATCOM

Systems:
```
strset - All, Netware
_strset - All, Netware
_fstrset - All
_wcsset - All
_mbsset - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsset - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <string.h>
size_t strspn( const char *str,
               const char *charset );
size_t _fstrspn( const char __far *str,
                 const char __far *charset );
#include <wchar.h>
size_t wcsspn( const wchar_t *str,
               const wchar_t *charset );
#include <wchar.h>
size_t _mbsspn( const unsigned char *str,
                const unsigned char *charset );
size_t _fmbsspn( const unsigned char __far *str,
                 const unsigned char __far *charset );
```

Description: The `strspn` function computes the length, in bytes, of the initial segment of the string pointed to by *str* which consists of characters from the string pointed to by *charset.* The terminating null character is not considered to be part of *charset.*

The `_fstrspn` function is a data model independent form of the `strspn` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `wcsspn` function is a wide-character version of `strspn` that operates with wide-character strings.

The `_mbsspn` function is a multibyte character version of `strspn` that operates with multibyte character strings.

The `_fmbsspn` function is a data model independent form of the `_mbsspn` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The length, in bytes, of the initial segment is returned.

See Also: `strcspn`, `strspnp`

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

void main()
  {
    printf( "%d\n", strspn( "out to lunch", "aeiou" ) );
    printf( "%d\n", strspn( "out to lunch", "xyz" ) );
  }
```

produces the following:

```
2
0
```

Classification: strspn is ANSI, _fstrspn is not ANSI, wcsspn is ANSI, _mbsspn is not ANSI, _fmbsspn is not ANSI

Systems:
```
strspn - All, Netware
_fstrspn - All
wcsspn - All
_mbsspn - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsspn - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
char *strspnp( const char *str,
               const char *charset );
char *_strspnp( const char *str,
                const char *charset );
char __far *_fstrspnp( const char __far *str,
                       const char __far *charset );
#include <tchar.h>
wchar_t *_wcsspnp( const wchar_t *str,
                   const wchar_t *charset );
#include <mbstring.h>
unsigned char *_mbsspnp( const unsigned char *str,
                         const unsigned char *charset );
unsigned char __far *_fmbsspnp(
                    const unsigned char __far *str,
                    const unsigned char __far *charset );
```

Description: The `strspnp` function returns a pointer to the first character in *str* that does not belong to the set of characters in *charset.* The terminating null character is not considered to be part of *charset.*

The `_strspnp` function is identical to `strspnp`. Use `_strspnp` for ANSI/ISO naming conventions.

The `_fstrspnp` function is a data model independent form of the `strspnp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The `_wcsspnp` function is a wide-character version of `strspnp` that operates with wide-character strings.

The `_mbsspnp` function is a multibyte character version of `strspnp` that operates with multibyte character strings.

The `_fmbsspnp` function is a data model independent form of the `_mbsspnp` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strspnp` function returns `NULL` if *str* consists entirely of characters from *charset.*

See Also: `strcspn`, `strspn`

Example:

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

void main()
  {
    printf( "%s\n", strspnp( "out to lunch", "aeiou" ) );
    printf( "%s\n", strspnp( "out to lunch", "xyz" ) );
  }
```

produces the following:

```
t to lunch
out to lunch
```

Classification: WATCOM

_strspnp conforms to ANSI/ISO naming conventions

Systems:

```
strspnp - All, Netware
_strspnp - All, Netware
_fstrspnp - All
_wcsspnp - All
_mbsspnp - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsspnp - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:

```
#include <string.h>
char *strstr( const char *str,
              const char *substr );
char __far *_fstrstr( const char __far *str,
                      const char __far *substr );
#include <wchar.h>
wchar_t *wcsstr( const wchar_t *str,
                 const wchar_t *substr );
#include <mbstring.h>
unsigned char *_mbsstr( const unsigned char *str,
                        const unsigned char *substr );
unsigned char __far *_fmbsstr(
                    const unsigned char __far *str,
                    const unsigned char __far *substr );
```

Description: The `strstr` function locates the first occurrence in the string pointed to by *str* of the sequence of characters (excluding the terminating null character) in the string pointed to by *substr.*

The `_fstrstr` function is a data model independent form of the `strstr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcsstr` function is a wide-character version of `strstr` that operates with wide-character strings.

The `_mbsstr` function is a multibyte character version of `strstr` that operates with multibyte character strings.

The `_fmbsstr` function is a data model independent form of the `_mbsstr` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strstr` function returns a pointer to the located string, or `NULL` if the string is not found.

See Also: `strcspn`

Example:

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

void main()
  {
    printf( "%s\n", strstr("This is an example", "is") );
  }
```

produces the following:

```
is is an example
```

Classification: strstr is ANSI, _fstrstr is not ANSI, wcsstr is ANSI, _mbsstr is not ANSI, _fmbsstr is not ANSI

Systems:
```
strstr - All, Netware
_fstrstr - All
wcsstr - All
_mbsstr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsstr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <time.h>
char *_strtime( char *timestr )
wchar_t _wstrtime( wchar_t *timestr );
```

Description: The `_strtime` function copies the current time to the buffer pointed to by *timestr*. The time is formatted as "HH:MM:SS" where "HH" is two digits representing the hour in 24-hour notation, where "MM" is two digits representing the minutes past the hour, and where "SS" is two digits representing seconds. The buffer must be at least 9 bytes long.

The `_wstrtime` function is a wide-character version of `_strtime` that operates with wide-character strings.

Returns: The `_strtime` function returns a pointer to the resulting text string *timestr*.

See Also: `asctime`, `ctime`, `gmtime`, `localtime`, `mktime`, `_strdate`, `time`, `tzset`

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

void main()
  {
    char timebuff[9];

    printf( "%s\n", _strtime( timebuff ) );
  }
```

Classification: WATCOM

Systems:
```
_strtime - All
_wstrtime - All
```

Synopsis:

```
#include <stdlib.h>
double strtod( const char *ptr, char **endptr );
#include <wchar.h>
double wcstod( const wchar_t *ptr, wchar_t **endptr );
```

Description: The `strtod` function converts the string pointed to by *ptr* to `double` representation. The function recognizes a string containing:

- optional white space,
- an optional plus or minus sign,
- a sequence of digits containing an optional decimal point,
- an optional 'e' or 'E' followed by an optionally signed sequence of digits.

The conversion ends at the first unrecognized character. A pointer to that character will be stored in the object to which *endptr* points if *endptr* is not `NULL.` By comparing the "end" pointer with *ptr*, it can be determined how much of the string, if any, was scanned by the `strtod` function.

The `wcstod` function is a wide-character version of `strtod` that operates with wide-character strings.

Returns: The `strtod` function returns the converted value. If the correct value would cause overflow, plus or minus `HUGE_VAL` is returned according to the sign, and `errno` is set to `ERANGE.` If the correct value would cause underflow, then zero is returned, and `errno` is set to `ERANGE.` Zero is returned when the input string cannot be converted. In this case, `errno` is not set. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `atof`

Example:

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

void main()
  {
    double pi;

    pi = strtod( "3.141592653589793", NULL );
    printf( "pi=%17.15f\n",pi );
  }
```

Classification: strtod is ANSI, wcstod is ANSI

Systems:
```
strtod - Math
wcstod - Math
```

Synopsis:

```
#include <string.h>
char *strtok( char *s1, const char *s2 );
char __far *_fstrtok( char __far *s1,
                      const char __far *s2 );
#include <wchar.h>
wchar_t *wcstok( wchar_t *s1, const wchar_t *s2,
                 wchar_t **ptr );
#include <mbstring.h>
unsigned char *_mbstok( unsigned char *s1,
                  const unsigned char *s2 );
unsigned char __far *_fmbstok( unsigned char __far *s1,
                         const unsigned char __far *s2 );
```

Description: The `strtok` function is used to break the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call to `strtok` will return a pointer to the first token in the string pointed to by *s1*. Subsequent calls to `strtok` must pass a NULL pointer as the first argument, in order to get the next token in the string. The set of delimiters used in each of these calls to `strtok` can be different from one call to the next.

The first call in the sequence searches *s1* for the first character that is not contained in the current delimiter string *s2*. If no such character is found, then there are no tokens in *s1* and the `strtok` function returns a NULL pointer. If such a character is found, it is the start of the first token.

The `strtok` function then searches from there for a character that is contained in the current delimiter string. If no such character is found, the current token extends to the end of the string pointed to by *s1*. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the following character, from which the next search for a token will start when the first argument is a NULL pointer.

Because `strtok` may modify the original string, that string should be duplicated if the string is to be re-used.

The `_fstrtok` function is a data model independent form of the `strtok` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `wcstok` function is a wide-character version of `strtok` that operates with wide-character strings. The third argument *ptr* points to a caller-provided `wchar_t` pointer into which the `wcstok` function stores information necessary for it to continue scanning the same wide string.

On the first call in the sequence of calls to `wcstok`, *s1* points to a wide string. In subsequent calls for the same string, *s1* must be NULL. If *s1* is NULL, the value pointed to by *ptr* matches that set by the previous call to `wcstok` for the same wide string. Otherwise, the value of *ptr* is ignored. The list of delimiters pointed to by *s2* may be different from one call to the next. The tokenization of *s1* is similar to that for the `strtok` function.

The `_mbstok` function is a multibyte character version of `strtok` that operates with multibyte character strings.

The `_fmbstok` function is a data model independent form of the `_mbstok` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: The `strtok` function returns a pointer to the first character of a token or `NULL` if there is no token found.

See Also: `strcspn`, `strpbrk`

Example:

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

void main()
  {
    char *p;
    char *buffer;
    char *delims = { " .," };

    buffer = strdup( "Find words, all of them." );
    printf( "%s\n", buffer );
    p = strtok( buffer, delims );
    while( p != NULL ) {
      printf( "word: %s\n", p );
      p = strtok( NULL, delims );
    }
    printf( "%s\n", buffer );
  }
```

produces the following:

```
Find words, all of them.
word: Find
word: words
word: all
word: of
word: them
Find
```

Classification: strtok is ANSI, _fstrtok is not ANSI, wcstok is ANSI, _mbstok is not ANSI, _fmbstok is not ANSI

Systems:
```
strtok - All, Netware
_fstrtok - All
wcstok - All
_mbstok - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbstok - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <stdlib.h>
long int strtol( const char *ptr,
                 char **endptr,
                 int base );
#include <wchar.h>
long int wcstol( const wchar_t *ptr,
                 wchar_t **endptr,
                 int base );
```

Description: The `strtol` function converts the string pointed to by *ptr* to an object of type `long int.` The `strtol` function recognizes a string containing:

- optional white space,
- an optional plus or minus sign,
- a sequence of digits and letters.

The conversion ends at the first unrecognized character. A pointer to that character will be stored in the object to which *endptr* points if *endptr* is not `NULL.`

If *base* is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are "0x" or "0X" the digits are treated as hexadecimal. If the first character is ’0’, the digits are treated as octal. Otherwise the digits are treated as decimal.

If *base* is not zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits.

The `wcstol` function is a wide-character version of `strtol` that operates with wide-character strings.

Returns: The `strtol` function returns the converted value. If the correct value would cause overflow, `LONG_MAX` or `LONG_MIN` is returned according to the sign, and `errno` is set to `ERANGE.` If *base* is out of range, zero is returned and `errno` is set to `EDOM.`

See Also: `atoi`, `atol`, `itoa`, `ltoa`, `sscanf`, `strtoul`, `ultoa`, `utoa`

Example:

```
#include <stdlib.h>

void main()
  {
    long int v;

    v = strtol( "12345678", NULL, 10 );
  }
```

Classification: strtol is ANSI, wcstol is ANSI

Systems:

```
strtol - All, Netware
wcstol - All
```

Synopsis:

```
#include <stdlib.h>
unsigned long int strtoul( const char *ptr,
                           char **endptr,
                           int base );
#include <wchar.h>
unsigned long int wcstoul( const wchar_t *ptr,
                           wchar_t **endptr,
                           int base );
```

Description: The `strtoul` function converts the string pointed to by *ptr* to an `unsigned long.` The function recognizes a string containing optional white space, an optional sign (+ or -), followed by a sequence of digits and letters. The conversion ends at the first unrecognized character. A pointer to that character will be stored in the object *endptr* points to if *endptr* is not `NULL.`

If *base* is zero, the first characters determine the base used for the conversion. If the first characters are "0x" or "0X" the digits are treated as hexadecimal. If the first character is ’0’, the digits are treated as octal. Otherwise the digits are treated as decimal.

If *base* is not zero, it must have a value of between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than *base* are permitted. If the value of *base* is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits.

If there is a leading minus sign in the string, the value is negated.

The `wcstoul` function is a wide-character version of `strtoul` that operates with wide-character strings.

Returns: The `strtoul` function returns the converted value. If the correct value would cause overflow, `ULONG_MAX` is returned and `errno` is set to `ERANGE.` If *base* is out of range, zero is returned and `errno` is set to `EDOM.`

See Also: `atoi`, `atol`, `itoa`, `ltoa`, `sscanf`, `strtol`, `ultoa`, `utoa`

Example:

```
#include <stdlib.h>

void main()
  {
    unsigned long int v;

    v = strtoul( "12345678", NULL, 10 );
  }
```

Classification: strtoul is ANSI, wcstoul is ANSI

Systems:
```
strtoul - All, Netware
wcstoul - All
```

Synopsis:

```
#include <string.h>
char *strupr( char *s );
char *_strupr( char *s );
char __far *_fstrupr( char __far *s );
#include <wchar.h>
wchar_t *_wcsupr( wchar_t *s );
#include <mbstring.h>
unsigned char *_mbsupr( unsigned char *s );
unsigned char __far *_fmbsupr( unsigned char __far *s );
```

Description: The `strupr` function replaces the string *s* with uppercase characters by invoking the `toupper` function for each character in the string.

The `_strupr` function is identical to `strupr`. Use `_strupr` for ANSI/ISO naming conventions.

The `_fstrupr` function is a data model independent form of the `strupr` function. It accepts far pointer arguments and returns a far pointer. It is most useful in mixed memory model applications.

The `_wcsupr` function is a wide-character version of `strupr` that operates with wide-character strings.

The `_mbsupr` function is a multibyte character version of `strupr` that operates with multibyte character strings.

Returns: The address of the original string *s* is returned.

See Also: `strlwr`

Example:

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

char source[] = { "A mixed-case STRING" };

void main()
  {
    printf( "%s\n", source );
    printf( "%s\n", strupr( source ) );
    printf( "%s\n", source );
  }
```

produces the following:

```
A mixed-case STRING
A MIXED-CASE STRING
A MIXED-CASE STRING
```

Classification: WATCOM

_strupr conforms to ANSI/ISO naming conventions

Systems:
```
strupr - All, Netware
_strupr - All, Netware
_fstrupr - All
_wcsupr - All
_mbsupr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fmbsupr - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <string.h>
size_t strxfrm( char *dst,
                const char *src,
                size_t n );
#include <wchar.h>
size_t wcsxfrm( wchar_t *dst,
                const wchar_t *src,
                size_t n );
```

Description: The `strxfrm` function transforms, for no more than *n* characters, the string pointed to by *src* to the buffer pointed to by *dst*. The transformation uses the collating sequence selected by the `setlocale` function so that two transformed strings will compare identically (using the `strncmp` function) to a comparison of the original two strings using the `strcoll` function. The function will be equivalent to the `strncpy` function (except there is no padding of the *dst* argument with null characters when the argument *src* is shorter than *n* characters) when the collating sequence is selected from the `"C"` locale.

The `wcsxfrm` function is a wide-character version of `strxfrm` that operates with wide-character strings. For `wcsxfrm`, after the string transformation, a call to `wcscmp` with the two transformed strings yields results identical to those of a call to `wcscoll` applied to the original two strings. `wcsxfrm` and `strxfrm` behave identically otherwise.

Returns: The `strxfrm` function returns the length of the transformed string. If this length is more than *n*, the contents of the array pointed to by *dst* are indeterminate.

See Also: `setlocale`, `strcoll`

Example:

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

char src[] = { "A sample STRING" };
char dst[20];

void main()
  {
    size_t len;

    setlocale( LC_ALL, "C" );
    printf( "%s\n", src );
    len = strxfrm( dst, src, 20 );
    printf( "%s (%u)\n", dst, len );
  }
```

produces the following:

```
A sample STRING
A sample STRING (15)
```

Classification: strxfrm is ANSI, wcsxfrm is ANSI

Systems:
```
strxfrm - All, Netware
wcsxfrm - All
```

Synopsis:

```
#include <stdlib.h>
void swab( char *src, char *dest, int num );
```

Description: The `swab` function copies *num* bytes (which should be even) from *src* to *dest* swapping every pair of characters. This is useful for preparing binary data to be transferred to another machine that has a different byte ordering.

Returns: The `swab` function has no return value.

Example:

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

char *msg = "hTsim seasegi  swspaep.d";
#define NBYTES 24

void main()
  {
    auto char buffer[80];

    printf( "%s\n", msg );
    memset( buffer, '\0', 80 );
    swab( msg, buffer, NBYTES );
    printf( "%s\n", buffer );
  }
```

produces the following:

```
hTsim seasegi  swspaep.d
This message is swapped.
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
int system( const char *command );
int _wsystem( const wchar_t *command );
```

Description: If the value of *command* is `NULL,` then the `system` function determines whether or not a command processor is present ("COMMAND.COM" in DOS and Windows 95/98 or "CMD.EXE" in OS/2 and Windows NT/2000).

Otherwise, the `system` function invokes a copy of the command processor, and passes the string *command* to it for processing. This function uses `spawnl` to load a copy of the command processor identified by the `COMSPEC` environment variable.

This means that any command that can be entered to DOS can be executed, including programs, DOS commands and batch files. The `exec...` and `spawn...` functions can only cause programs to be executed.

The `_wsystem` function is identical to `system` except that it accepts a wide-character string argument.

Returns: If the value of *command* is `NULL,` then the `system` function returns zero if the command processor is not present, a non-zero value if the command processor is present. Note that Microsoft Windows 3.x does not support a command shell and so the `system` function always returns zero when *command* is `NULL.`

Otherwise, the `system` function returns the result of invoking a copy of the command processor. A non-zero value is returned if the command processor could not be loaded; otherwise, zero is returned. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `abort`, `atexit`, `_bgetcmd`, `exec` Functions, `exit`, `_exit`, `getcmd`, `getenv`, `main`, `onexit`, `putenv`, `spawn` Functions

Example:

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

void main()
  {
    int rc;
```

```
        rc = system( "dir" );
        if( rc != 0 ) {
          printf( "shell could not be run\n" );
        }
      }
```

Classification: system is ANSI, POSIX 1003.2, _wsystem is not ANSI

Systems:
```
system - All, Netware
_wsystem - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

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

Description: The `tan` function computes the tangent of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns: The `tan` function returns the tangent value. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `atan`, `atan2`, `cos`, `sin`, `tanh`

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

void main()
  {
    printf( "%f\n", tan(.5) );
  }
```

produces the following:

```
0.546302
```

Classification: ANSI

Systems: Math

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

Description: The `tanh` function computes the hyperbolic tangent of *x*.

When the *x* argument is large, partial or total loss of significance may occur. The `matherr` function will be invoked in this case.

Returns: The `tanh` function returns the hyperbolic tangent value. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `cosh`, `sinh`, `matherr`

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

void main()
  {
    printf( "%f\n", tanh(.5) );
  }
```

produces the following:

```
0.462117
```

Classification: ANSI

Systems: Math

Synopsis:

```
#include <io.h>
long tell( int handle );
__int64 _telli64( int handle );
```

Description: The `tell` function reports the current file position at the operating system level. The *handle* value is the file handle returned by a successful execution of the `open` function.

The returned value may be used in conjunction with the `lseek` function to reset the current file position.

The _ `telli`64 function is similar to the `tell` function but returns a 64-bit file position. This value may be used in conjunction with the `_lseeki64` function to reset the current file position.

Returns: If an error occurs in `tell`, (-1L) is returned.

If an error occurs in _ `telli`64, (-1I64) is returned.

When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Otherwise, the current file position is returned in a system-dependent manner. A value of 0 indicates the start of the file.

See Also: `chsize`, `close`, `creat`, `dup`, `dup2`, `eof`, `exec` Functions, `fdopen`, `filelength`, `fileno`, `fstat`, `_grow_handles`, `isatty`, `lseek`, `open`, `read`, `setmode`, `sopen`, `stat`, `write`, `umask`

Example:

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

char buffer[]
        = { "A text record to be written" };
```

```
void main()
  {
    int handle;
    int size_written;

    /* open a file for output             */
    /* replace existing file if it exists */
    handle = open( "file",
                O_WRONLY | O_CREAT | O_TRUNC | O_TEXT,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    if( handle != -1 ) {

      /* print file position */
      printf( "%ld\n", tell( handle ) );

      /* write the text */
      size_written = write( handle, buffer,
                            sizeof( buffer ) );

      /* print file position */
      printf( "%ld\n", tell( handle ) );

      /* close the file */
      close( handle );
    }
  }
```

produces the following:

```
0
28
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <stdio.h>
char *_tempnam( char *dir, char *prefix );
wchar_t *_wtempnam( wchar_t *dir, wchar_t *prefix );
```

Description: `_tempnam` creates a temporary filename for use in another directory. This filename is different from that of any existing file. The *prefix* argument is the prefix to the filename. `_tempnam` uses `malloc` to allocate space for the filename; the program is responsible for freeing this space when it is no longer needed. `_tempnam` looks for the file with the given name in the following directories, listed in order of precedence.

Directory Used Conditions

Directory specified by TMP The `TMP` environment variable must be set and the directory specified by `TMP` must exist.

dir (function argument) The `TMP` environment variable must not be set or the directory specified by `TMP` does not exist.

_P_tmpdir (_wP_tmpdir) in STDIO.H The *dir* argument is NULL or *dir* is the name of a nonexistent directory. The `_wP_tmpdir` string is used by `_wtempnam`.

Current working directory `_tempnam` uses the current working directory when `_P_tmpdir` does not exist. `_wtempnam` uses the current working directory when `_wP_tmpdir` does not exist.

`_tempnam` automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the OEM code page obtained from the operating system. `_wtempnam` is a wide-character version of `_tempnam` the arguments and return value of `_wtempnam` are wide-character strings. `_wtempnam` and `_tempnam` behave identically except that `_wtempnam` does not handle multibyte-character strings.

The function generates unique filenames for up to `TMP_MAX` calls.

Returns: The `_tempnam` function returns a pointer to the name generated, unless it is impossible to create this name or the name is not unique. If the name cannot be created or if a file with that name already exists, `_tempnam` returns NULL.

See Also: `fopen`, `freopen`, `_mktemp`, `tmpfile`, `tmpnam`

Example:

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

/*
  Environment variable TMP=C:\WINDOWS\TEMP
*/
void main()
  {
    char *filename;

    FILE *fp;

    filename = _tempnam( "D:\\TEMP", "_T" );
    if( filename == NULL )
        printf( "Can’t obtain temp file name\n" );
    else {
        printf( "Temp file name is %s\n", filename );
        fp = fopen( filename, "w+b" );
        /* . */
        /* . */
        /* . */
        fclose( fp );
        remove( filename );
        free( filename );
    }
  }
```

produces the following:

```
Temp file name is C:\WINDOWS\TEMP\_T1
```

Classification: WATCOM

Systems:

```
_tempnam - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wtempnam - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <time.h>
time_t time( time_t *tloc );
```

Description: The `time` function determines the current calendar time and encodes it into the type `time_t`.

The time represents the time since January 1, 1970 Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

The time set on the computer with the DOS `time` command and the DOS `date` command reflects the local time. The environment variable `TZ` is used to establish the time zone to which this local time applies. See the section *The TZ Environment Variable* for a discussion of how to set the time zone.

Returns: The `time` function returns the current calendar time. If *tloc* is not `NULL`, the current calendar time is also stored in the object pointed to by *tloc*.

See Also: `asctime`, `clock`, `ctime`, `difftime`, `gmtime`, `localtime`, `mktime`, `strftime`, `tzset`

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

void main()
  {
    time_t time_of_day;

    time_of_day = time( NULL );
    printf( "It is now: %s", ctime( &time_of_day ) );
  }
```

produces the following:

```
It is now: Fri Dec 25 15:58:42 1987
```

Classification: ANSI, POSIX 1003.1

Systems: All, Netware

Synopsis:
```
#include <stdio.h>
FILE *tmpfile( void );
```

Description: The `tmpfile` function creates a temporary binary file that will automatically be removed when it is closed or at program termination. The file is opened for update. For all systems except NetWare, the temporary file is located in the path specified by one of the following environment variables, if one is defined. Otherwise, the current working directory is used. They are listed in the order examined: `TMP`, `TEMP`, `TMPDIR`, and `TEMPDIR`.

Returns: The `tmpfile` function returns a pointer to the stream of the file that it created. If the file cannot be created, the `tmpfile` function returns `NULL`. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `fopen`, `freopen`, `_mktemp`, `_tempnam`, `tmpnam`

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

static FILE *TempFile;

void main()
  {
    TempFile = tmpfile();
    /* . */
    /* . */
    /* . */
    fclose( TempFile );
  }
```

Classification: ANSI

Systems: All, Netware

Synopsis:
```
#include <stdio.h>
char *tmpnam( char *buffer );
wchar_t *_wtmpnam( wchar_t *buffer );
```

Description: The `tmpnam` function generates a unique string for use as a valid file name. The `_wtmpnam` function is identical to `tmpnam` except that it generates a unique wide-character string for the file name. An internal static buffer is used to construct the filename. Subsequent calls to `tmpnam` reuse the internal buffer.

The function generates unique filenames for up to `TMP_MAX` calls.

Returns: If the argument *buffer* is a NULL pointer, `tmpnam` returns a pointer to an internal buffer containing the temporary file name. If the argument *buffer* is not a NULL pointer, `tmpnam` copies the temporary file name from the internal buffer to the specified buffer and returns a pointer to the specified buffer. It is assumed that the specified buffer is an array of at least `L_tmpnam` characters.

If the argument *buffer* is a NULL pointer, you may wish to duplicate the resulting string since subsequent calls to `tmpnam` reuse the internal buffer.

```
    char *name1, *name2;

    name1 = strdup( tmpnam( NULL ) );
    name2 = strdup( tmpnam( NULL ) );
```

See Also: `fopen`, `freopen`, `_mktemp`, `_tempnam`, `tmpfile`

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

void main()
  {
    char filename[ L_tmpnam ];
    FILE *fp;

    tmpnam( filename );
    fp = fopen( filename, "w+b" );
    /* . */
    /* . */
    /* . */
    fclose( fp );
    remove( filename );
  }
```

Classification: tmpnam is ANSI, _wtmpnam is not ANSI

Systems:
```
tmpnam - All, Netware
_wtmpnam - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <ctype.h>
int tolower( int c );
int _tolower( int c );
#include <wchar.h>
wint_t towlower( wint_t c );
```

Description: The `tolower` function converts *c* to a lowercase letter if *c* represents an uppercase letter.

The `_tolower` function is a version of `tolower` to be used only when *c* is known to be uppercase.

The `towlower` function is similar to `tolower` except that it accepts a wide-character argument.

Returns: The `tolower` function returns the corresponding lowercase letter when the argument is an uppercase letter; otherwise, the original character is returned. The `towlower` function returns the corresponding wide-character lowercase letter when the argument is a wide-character uppercase letter; otherwise, the original wide character is returned.

The result of `_tolower` is undefined if *c* is not an uppercase letter.

See Also: `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `iswctype`, `isxdigit`, `strlwr`, `strupr`, `toupper`

Example:

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

char chars[] = {
    'A',
    '5',
    '$',
    'Z'
};

#define SIZE sizeof( chars ) / sizeof( char )

void main()
  {
    int   i;
    for( i = 0; i < SIZE; i++ ) {
        printf( "%c ", tolower( chars[ i ] ) );
    }
    printf( "\n" );
  }
```

produces the following:

```
a 5 $ z
```

Classification: tolower is ANSI, _tolower is not ANSI, towlower is ANSI

Systems:

```
tolower - All, Netware
_tolower - All, Netware
towlower - All, Netware
```

Synopsis:

```
#include <ctype.h>
int toupper( int c );
int _toupper( int c );
#include <wchar.h>
wint_t towupper( wint_t c );
```

Description: The `toupper` function converts *c* to a uppercase letter if *c* represents a lowercase letter.

The `_toupper` function is a version of `toupper` to be used only when *c* is known to be lowercase.

The `towupper` function is similar to `toupper` except that it accepts a wide-character argument.

Returns: The `toupper` function returns the corresponding uppercase letter when the argument is a lowercase letter; otherwise, the original character is returned. The `towupper` function returns the corresponding wide-character uppercase letter when the argument is a wide-character lowercase letter; otherwise, the original wide character is returned.

The result of `_toupper` is undefined if *c* is not a lowercase letter.

See Also: `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `iswctype`, `isxdigit`, `strlwr`, `strupr`, `tolower`

Example:

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

char chars[] = {
    'a',
    '5',
    '$',
    'z'
};

#define SIZE sizeof( chars ) / sizeof( char )

void main()
  {
    int   i;
    for( i = 0; i < SIZE; i++ ) {
        printf( "%c ", toupper( chars[ i ] ) );
    }
    printf( "\n" );
  }
```

produces the following:

```
A 5 $ Z
```

Classification: toupper is ANSI, _toupper is not ANSI, towupper is ANSI

Systems:
```
toupper - All, Netware
_toupper - All, Netware
towupper - All, Netware
```

Synopsis:
```
#include <time.h>
void tzset( void );
```

Description: The `tzset` function sets the global variables `daylight`, `timezone` and `tzname` according to the value of the `TZ` environment variable. The section *The TZ Environment Variable* describes how to set this variable.

Under Win32, `tzset` also uses operating system supplied time zone information. The `TZ` environment variable can be used to override this information.

The global variables have the following values after `tzset` is executed:

daylight	Zero indicates that daylight saving time is not supported in the locale; a non-zero value indicates that daylight saving time is supported in the locale. This variable is cleared/set after a call to the `tzset` function depending on whether a daylight saving time abbreviation is specified in the `TZ` environment variable.
timezone	Contains the number of seconds that the local time zone is earlier than Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).
tzname	Two-element array pointing to strings giving the abbreviations for the name of the time zone when standard and daylight saving time are in effect.

The time set on the computer with the DOS `time` command and the DOS `date` command reflects the local time. The environment variable `TZ` is used to establish the time zone to which this local time applies. See the section *The TZ Environment Variable* for a discussion of how to set the time zone.

Returns: The `tzset` function does not return a value.

See Also: `ctime`, `localtime`, `mktime`, `strftime`

Example:

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

void print_zone()
  {
    char *tz;

    printf( "TZ: %s\n", (tz = getenv( "TZ" ))
                    ? tz : "default EST5EDT" );
    printf( "  daylight: %d\n", daylight );
    printf( "  timezone: %ld\n", timezone );
    printf( "  time zone names: %s %s\n",
            tzname[0], tzname[1] );
  }

void main()
  {
    print_zone();
    setenv( "TZ", "PST8PDT", 1 );
    tzset();
    print_zone();
  }
```

produces the following:

```
TZ: default EST5EDT
  daylight: 1
  timezone: 18000
  time zone names: EST EDT
TZ: PST8PDT
  daylight: 1
  timezone: 28800
  time zone names: PST PDT
```

Classification: POSIX 1003.1

Systems: All, Netware

Synopsis:

```
#include <stdlib.h>
char *ultoa( unsigned long int value,
             char *buffer,
             int radix );
char *_ultoa( unsigned long int value,
              char *buffer,
              int radix );
wchar_t *_ultow( unsigned long int value,
                 wchar_t *buffer,
                 int radix );
wchar_t *_ultou( unsigned long int value,
                 wchar_t *buffer,
                 int radix );
```

Description: The `ultoa` function converts the unsigned binary integer *value* into the equivalent string in base *radix* notation storing the result in the character array pointed to by *buffer*. A null character is appended to the result. The size of *buffer* must be at least 33 bytes when converting values in base 2. The value of *radix* must satisfy the condition:

```
2 <= radix <= 36
```

The `_ultoa` function is identical to `ultoa`. Use `_ultoa` for ANSI/ISO naming conventions.

The `_ultow` function is identical to `ultoa` except that it produces a wide-character string (which is twice as long).

The `_ultow` Unicode function is identical to `ultoa` except that it produces a Unicode character string (which is twice as long).

Returns: The `ultoa` function returns the pointer to the result.

See Also: `atoi`, `atol`, `itoa`, `ltoa`, `sscanf`, `strtol`, `strtoul`, `utoa`

Example:

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

void print_value( unsigned long int value )
  {
    int base;
    char buffer[33];

    for( base = 2; base <= 16; base = base + 2 )
      printf( "%2d %s\n", base,
              ultoa( value, buffer, base ) );
  }

void main()
  {
    print_value( (unsigned) 12765L );
  }
```

produces the following:

```
 2 11000111011101
 4 3013131
 6 135033
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification: WATCOM

_ultoa conforms to ANSI/ISO naming conventions

Systems:
```
ultoa - All, Netware
_ultoa - All, Netware
_ultow - All
```

Synopsis:

```
#include <sys\types.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <io.h>
int umask( int cmask );
```

Description: The `umask` function sets the process's file mode creation mask to *cmask.* The process's file mode creation mask is used during `creat`, `open` or `sopen` to turn off permission bits in the *permission* argument supplied. In other words, if a bit in the mask is on, then the corresponding bit in the file's requested permission value is disallowed.

The argument *cmask* is a constant expression involving the constants described below. The access permissions for the file or directory are specified as a combination of bits (defined in the `<sys\stat.h>` header file).

The following bits define permissions for the owner.

Permission	***Meaning***
S_IRWXU	Read, write, execute/search
S_IRUSR	Read permission
S_IWUSR	Write permission
S_IXUSR	Execute/search permission

The following bits define permissions for the group.

Permission	***Meaning***
S_IRWXG	Read, write, execute/search
S_IRGRP	Read permission
S_IWGRP	Write permission
S_IXGRP	Execute/search permission

The following bits define permissions for others.

Permission	***Meaning***
S_IRWXO	Read, write, execute/search
S_IROTH	Read permission
S_IWOTH	Write permission
S_IXOTH	Execute/search permission

The following bits define miscellaneous permissions used by other implementations.

Permission	***Meaning***
S_IREAD	is equivalent to S_IRUSR (read permission)
S_IWRITE	is equivalent to S_IWUSR (write permission)
S_IEXEC	is equivalent to S_IXUSR (execute/search permission)

For example, if `S_IRUSR` is specified, then reading is not allowed (i.e., the file is write only). If `S_IWUSR` is specified, then writing is not allowed (i.e., the file is read only).

Returns: The `umask` function returns the previous value of *cmask*.

See Also: `chmod`, `creat`, `mkdir`, `open`, `sopen`

Example:

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

void main()
  {
    int old_mask;

    /* set mask to create read-only files */
    old_mask = umask( S_IWUSR | S_IWGRP | S_IWOTH |
                      S_IXUSR | S_IXGRP | S_IXOTH );
  }
```

Classification: POSIX 1003.1

Systems: All, Netware

Synopsis:

```
#include <stdio.h>
int ungetc( int c, FILE *fp );
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc( wint_t c, FILE *fp );
```

Description: The `ungetc` function pushes the character specified by *c* back onto the input stream pointed to by *fp*. This character will be returned by the next read on the stream. The pushed-back character will be discarded if a call is made to the `fflush` function or to a file positioning function (`fseek`, `fsetpos` or `rewind`) before the next read operation is performed.

Only one character (the most recent one) of pushback is remembered.

The `ungetc` function clears the end-of-file indicator, unless the value of *c* is `EOF.`

The `ungetwc` function is identical to `ungetc` except that it pushes the wide character specified by *c* back onto the input stream pointed to by *fp*.

The `ungetwc` function clears the end-of-file indicator, unless the value of *c* is `WEOF.`

Returns: The `ungetc` function returns the character pushed back.

See Also: `fgetc`, `fgetchar`, `fgets`, `fopen`, `getc`, `getchar`, `gets`

Example:

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

void main()
  {
    FILE *fp;
    int c;
    long value;

    fp = fopen( "file", "r" );
    value = 0;
    c = fgetc( fp );
    while( isdigit(c) ) {
        value = value*10 + c - '0';
        c = fgetc( fp );
    }
    ungetc( c, fp ); /* put last character back */
    printf( "Value=%ld\n", value );
    fclose( fp );
  }
```

Classification: ungetc is ANSI, ungetwc is ANSI

Systems:
```
ungetc - All, Netware
ungetwc - All
```

Synopsis:

```
#include <conio.h>
int ungetch( int c );
```

Description: The `ungetch` function pushes the character specified by *c* back onto the input stream for the console. This character will be returned by the next read from the console (with `getch` or `getche` functions) and will be detected by the function `kbhit.` Only the last character returned in this way is remembered.

The `ungetch` function clears the end-of-file indicator, unless the value of *c* is `EOF.`

Returns: The `ungetch` function returns the character pushed back.

See Also: `getch`, `getche`, `kbhit`, `putch`

Example:

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

void main()
  {
    int c;
    long value;

    value = 0;
    c = getche();
    while( isdigit( c ) ) {
        value = value*10 + c - '0';
        c = getche();
    }
    ungetch( c );
    printf( "Value=%ld\n", value );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <io.h>
int unlink( const char *path );
int _wunlink( const wchar_t *path );
```

Description: The `unlink` function deletes the file whose name is the string pointed to by *path*. This function is equivalent to the `remove` function.

The `_wunlink` function is identical to `unlink` except that it accepts a wide-character string argument.

Returns: The `unlink` function returns zero if the operation succeeds, non-zero if it fails.

See Also: `chdir`, `chmod`, `close`, `getcwd`, `mkdir`, `open`, `remove`, `rename`, `rmdir`, `stat`

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

void main()
  {
    unlink( "vm.tmp" );
  }
```

Classification: unlink is POSIX 1003.1, _wunlink is not POSIX

Systems:
```
unlink - All, Netware
_wunlink - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:
```
#include <io.h>
int unlock( int handle,
            unsigned long offset,
            unsigned long nbytes );
```

Description: The `unlock` function unlocks *nbytes* amount of previously locked data in the file designated by *handle* starting at byte *offset* in the file. This allows other processes to lock this region of the file.

Multiple regions of a file can be locked, but no overlapping regions are allowed. You cannot unlock multiple regions in the same call, even if the regions are contiguous. All locked regions of a file should be unlocked before closing a file or exiting the program.

With DOS, locking is supported by version 3.0 or later. Note that `SHARE.COM` or `SHARE.EXE` must be installed.

Returns: The `unlock` function returns zero if successful, and -1 when an error occurs. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `lock`, `locking`, `open`, `sopen`

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

void main()
  {
    int handle;
    char buffer[20];

    handle = open( "file", O_RDWR | O_TEXT );
    if( handle != -1 ) {
      if( lock( handle, 0L, 20L ) ) {
        printf( "Lock failed\n" );
      } else {
        read( handle, buffer, 20 );
        /* update the buffer here */
        lseek( handle, 0L, SEEK_SET );
        write( handle, buffer, 20 );
        unlock( handle, 0L, 20L );
      }
      close( handle );
    }
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <graph.h>
void _FAR _unregisterfonts( void );
```

Description: The `_unregisterfonts` function frees the memory previously allocated by the `_registerfonts` function. The currently selected font is also unloaded.

Attempting to use the `_setfont` function after calling `_unregisterfonts` will result in an error.

Returns: The `_unregisterfonts` function does not return a value.

See Also: `_registerfonts`, `_setfont`, `_getfontinfo`, `_outgtext`, `_getgtextextent`, `_setgtextvector`, `_getgtextvector`

Example:
```
#include <conio.h>
#include <stdio.h>
#include <graph.h>

main()
{
    int i, n;
    char buf[ 10 ];

    _setvideomode( _VRES16COLOR );
    n = _registerfonts( "*.fon" );
    for( i = 0; i < n; ++i ) {
        sprintf( buf, "n%d", i );
        _setfont( buf );
        _moveto( 100, 100 );
        _outgtext( "WATCOM Graphics" );
        getch();
        _clearscreen( _GCLEARSCREEN );
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

utime, _utime, _wutime

Synopsis:

```
#include <sys\utime.h>
int utime( const char *path,
           const struct utimbuf *times );
int _utime( const char *path,
           const struct utimbuf *times );
int _wutime( const wchar_t *path,
             const struct utimbuf *times );

struct utimbuf {
    time_t   actime;    /* access time */
    time_t   modtime;   /* modification time */
};
```

Description: The `utime` function records the access and modification times for the file identified by *path.*

The `_utime` function is identical to `utime`. Use `_utime` for ANSI naming conventions.

If the *times* argument is `NULL`, the access and modification times of the file or directory are set to the current time. Write access to this file must be permitted for the time to be recorded.

If the *times* argument is not `NULL`, it is interpreted as a pointer to a `utimbuf` structure and the access and modification times of the file or directory are set to the values contained in the designated structure. The access and modification times are taken from the `actime` and `modtime` fields in this structure.

The `_wutime` function is identical to `utime` except that *path* points to a wide-character string.

Returns: The `utime` function returns zero when the time was successfully recorded. A value of -1 indicates an error occurred.

Errors: When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

Constant	***Meaning***
EACCES	Search permission is denied for a component of *path* or the *times* argument is `NULL` and the effective user ID of the process does not match the owner of the file and write access is denied.
EINVAL	The date is before 1980 (DOS only).

EMFILE	There are too many open files.
ENOENT	The specified *path* does not exist or *path* is an empty string.

Example:

```
#include <stdio.h>
#include <sys\utime.h>

void main( int argc, char *argv[] )
  {
    if( (utime( argv[1], NULL ) != 0) && (argc > 1) ) {
       printf( "Unable to set time for %s\n", argv[1] );
    }
  }
```

Classification: utime is POSIX 1003.1, _utime is not POSIX, _wutime is not POSIX

Systems:

```
utime - All, Netware
_utime - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_wutime - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <stdlib.h>
char *utoa( unsigned int value,
            char *buffer,
            int radix );
char *_utoa( unsigned int value,
             char *buffer,
             int radix );
wchar_t *_utow( unsigned int value,
                wchar_t *buffer,
                int radix );
wchar_t *_utou( unsigned int value,
                wchar_t *buffer,
                int radix );
```

Description: The `utoa` function converts the unsigned binary integer *value* into the equivalent string in base *radix* notation storing the result in the character array pointed to by *buffer*. A null character is appended to the result. The size of *buffer* must be at least (8 * sizeof(int) + 1) bytes when converting values in base 2. That makes the size 17 bytes on 16-bit machines, and 33 bytes on 32-bit machines. The value of *radix* must satisfy the condition:

```
2 <= radix <= 36
```

The `_utoa` function is identical to `utoa`. Use `_utoa` for ANSI/ISO naming conventions.

The `_utow` function is identical to `utoa` except that it produces a wide-character string (which is twice as long).

The `_utow` Unicode function is identical to `utoa` except that it produces a Unicode character string (which is twice as long).

Returns: The `utoa` function returns the pointer to the result.

See Also: `atoi`, `atol`, `itoa`, `ltoa`, `sscanf`, `strtol`, `strtoul`, `ultoa`

Example:

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

void main()
  {
    int base;
    char buffer[18];
```

```
        for( base = 2; base <= 16; base = base + 2 )
          printf( "%2d %s\n", base,
                  utoa( (unsigned) 12765, buffer, base ) );
      }
```

produces the following:

```
 2 11000111011101
 4 3013131
 6 135033
 8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification: WATCOM

_utoa conforms to ANSI/ISO naming conventions

Systems:
```
utoa - All, Netware
_utoa - All, Netware
_utow - All
```

Synopsis:

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

Description: `va_arg` is a macro that can be used to obtain the next argument in a list of variable arguments. It must be used with the associated macros `va_start` and `va_end.` A sequence such as

```
    void example( char *dst, ... )
    {
        va_list curr_arg;
        int next_arg;

        va_start( curr_arg, dst );
        next_arg = va_arg( curr_arg, int );
        .
        .
        .
```

causes `next_arg` to be assigned the value of the next variable argument. The argument *type* (which is `int` in the example) is the type of the argument originally passed to the function.

The macro `va_start` must be executed first in order to properly initialize the variable `curr_arg` and the macro `va_end` should be executed after all arguments have been obtained.

The data item `curr_arg` is of type `va_list` which contains the information to permit successive acquisitions of the arguments.

Returns: The macro returns the value of the next variable argument, according to type passed as the second parameter.

See Also: `va_end`, `va_start`, `vfprintf`, `vprintf`, `vsprintf`

Example:

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

void test_fn( const char *msg,
              const char *types,
              ... );
```

```
void main()
  {
    printf( "VA...TEST\n" );
    test_fn( "PARAMETERS: 1, \"abc\", 546",
             "isi", 1, "abc", 546 );
    test_fn( "PARAMETERS: \"def\", 789",
             "si", "def", 789 );
  }

static void test_fn(
  const char *msg,   /* message to be printed    */
  const char *types, /* parameter types (i,s)    */
  ... )              /* variable arguments       */
  {
    va_list argument;
    int   arg_int;
    char *arg_string;
    const char *types_ptr;

    types_ptr = types;
    printf( "\n%s -- %s\n", msg, types );
    va_start( argument, types );
    while( *types_ptr != '\0' ) {
      if (*types_ptr == 'i') {
        arg_int = va_arg( argument, int );
        printf( "integer: %d\n", arg_int );
      } else if (*types_ptr == 's') {
        arg_string = va_arg( argument, char * );
        printf( "string:  %s\n", arg_string );
      }
      ++types_ptr;
    }
    va_end( argument );
  }
```

produces the following:

```
VA...TEST

PARAMETERS: 1, "abc", 546 -- isi
integer: 1
string:  abc
integer: 546

PARAMETERS: "def", 789 -- si
string:  def
integer: 789
```

Classification: ANSI

Systems: MACRO

Synopsis:
```
#include <stdarg.h>
void va_end( va_list param );
```

Description: `va_end` is a macro used to complete the acquisition of arguments from a list of variable arguments. It must be used with the associated macros `va_start` and `va_arg.` See the description for `va_arg` for complete documentation on these macros.

Returns: The macro does not return a value.

See Also: `va_arg`, `va_start`, `vfprintf`, `vprintf`, `vsprintf`

Example:
```
#include <stdio.h>
#include <stdarg.h>
#include <time.h>

#define ESCAPE 27

void tprintf( int row, int col, char *fmt, ... )
 {
    auto va_list ap;
    char *p1, *p2;

    va_start( ap, fmt );
    p1 = va_arg( ap, char * );
    p2 = va_arg( ap, char * );
    printf( "%c[%2.2d;%2.2dH", ESCAPE, row, col );
    printf( fmt, p1, p2 );
    va_end( ap );
 }

void main()
  {
    struct tm  time_of_day;
    time_t     ltime;
    auto char  buf[26];

    time( &ltime );
    _localtime( &ltime, &time_of_day );
    tprintf( 12, 1, "Date and time is: %s\n",
            _asctime( &time_of_day, buf ) );
  }
```

Classification: ANSI

Systems: MACRO

Synopsis:

```
#include <stdarg.h>
void va_start( va_list param, previous );
```

Description: `va_start` is a macro used to start the acquisition of arguments from a list of variable arguments. The *param* argument is used by the `va_arg` macro to locate the current acquired argument. The *previous* argument is the argument that immediately precedes the `"..."` notation in the original function definition. It must be used with the associated macros `va_arg` and `va_end.` See the description of `va_arg` for complete documentation on these macros.

Returns: The macro does not return a value.

See Also: `va_arg`, `va_end`, `vfprintf`, `vprintf`, `vsprintf`

Example:

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

#define ESCAPE 27

void tprintf( int row, int col, char *fmt, ... )
 {
    auto va_list ap;
    char *p1, *p2;

    va_start( ap, fmt );
    p1 = va_arg( ap, char * );
    p2 = va_arg( ap, char * );
    printf( "%c[%2.2d;%2.2dH", ESCAPE, row, col );
    printf( fmt, p1, p2 );
    va_end( ap );
 }

void main()
  {
    struct tm  time_of_day;
    time_t     ltime;
    auto char  buf[26];

    time( &ltime );
    _localtime( &ltime, &time_of_day );
    tprintf( 12, 1, "Date and time is: %s\n",
            _asctime( &time_of_day, buf ) );
  }
```

Classification: ANSI

Systems: MACRO

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>
int _vbprintf( char *buf, size_t bufsize,
               const char *format, va_list arg );
int _vbwprintf( wchar_t *buf, size_t bufsize,
                const wchar_t *format, va_list arg );
```

Description: The `_vbprintf` function formats data under control of the *format* control string and writes the result to *buf*. The argument *bufsize* specifies the size of the character array *buf* into which the generated output is placed. The *format* string is described under the description of the `printf` function. The `_vbprintf` function is equivalent to the `_bprintf` function, with the variable argument list replaced with *arg*, which has been initialized by the `va_start` macro.

The `_vbwprintf` function is identical to `_vbprintf` except that it accepts a wide-character string argument for *format* and produces wide-character output.

Returns: The `_vbprintf` function returns the number of characters written, or a negative value if an output error occurred.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `vcprintf`, `vfprintf`, `vprintf`, `vsprintf`

Example: The following shows the use of `_vbprintf` in a general error message routine.

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

char msgbuf[80];

char *fmtmsg( char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    strcpy( msgbuf, "Error: " );
    _vbprintf( &msgbuf[7], 73, format, arglist );
    va_end( arglist );
    return( msgbuf );
  }
```

```
void main()
  {
    char *msg;

    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );
  }
```

Classification: WATCOM

Systems:
```
_vbprintf - All, Netware
_vbwprintf - All
```

Synopsis:

```
#include <conio.h>
#include <stdarg.h>
int vcprintf( const char *format, va_list arg );
```

Description: The `vcprintf` function writes output directly to the console under control of the argument *format*. The `putch` function is used to output characters to the console. The *format* string is described under the description of the `printf` function. The `vcprintf` function is equivalent to the `cprintf` function, with the variable argument list replaced with *arg*, which has been initialized by the `va_start` macro.

Returns: The `vcprintf` function returns the number of characters written, or a negative value if an output error occurred. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `_vbprintf`, `vfprintf`, `vprintf`, `vsprintf`

Example:

```
#include <conio.h>
#include <stdarg.h>
#include <time.h>

#define ESCAPE 27

void tprintf( int row, int col, char *format, ... )
 {
    auto va_list arglist;

    cprintf( "%c[%2.2d;%2.2dH", ESCAPE, row, col );
    va_start( arglist, format );
    vcprintf( format, arglist );
    va_end( arglist );
 }

void main()
  {
    struct tm  time_of_day;
    time_t     ltime;
    auto char  buf[26];

    time( &ltime );
    _localtime( &ltime, &time_of_day );
    tprintf( 12, 1, "Date and time is: %s\n",
            _asctime( &time_of_day, buf ) );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:

```
#include <conio.h>
#include <stdarg.h>
int vcscanf( const char *format, va_list args )
```

Description: The `vcscanf` function scans input from the console under control of the argument *format.* The `vcscanf` function uses the function `getche` to read characters from the console. The *format* string is described under the description of the `scanf` function.

The `vcscanf` function is equivalent to the `cscanf` function, with a variable argument list replaced with *arg,* which has been initialized using the `va_start` macro.

Returns: The `vcscanf` function returns `EOF` when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned. When a file input error occurs, the `errno` global variable may be set.

See Also: `cscanf`, `fscanf`, `scanf`, `sscanf`, `va_arg`, `va_end`, `va_start`, `vfscanf`, `vscanf`, `vsscanf`

Example:

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

void cfind( char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    vcscanf( format, arglist );
    va_end( arglist );
  }

void main()
  {
    int day, year;
    char weekday[10], month[10];

    cfind( "%s %s %d %d",
            weekday, month, &day, &year );
    cprintf( "\n%s, %s %d, %d\n",
            weekday, month, day, year );
  }
```

Classification: WATCOM

Systems: All, Netware

Synopsis:
```
#include <stdarg.h>
#include <stdio.h>
int vfprintf( FILE *fp,
              const char *format,
              va_list arg );
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf( FILE *fp,
               const wchar_t *format,
               va_list arg );
```

Description: The `vfprintf` function writes output to the file pointed to by *fp* under control of the argument *format*. The *format* string is described under the description of the `printf` function. The `vfprintf` function is equivalent to the `fprintf` function, with the variable argument list replaced with *arg*, which has been initialized by the `va_start` macro.

The `vfwprintf` function is identical to `vfprintf` except that it accepts a wide-character string argument for *format*.

Returns: The `vfprintf` function returns the number of characters written, or a negative value if an output error occurred. The `vfwprintf` function returns the number of wide characters written, or a negative value if an output error occurred. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `_vbprintf`, `vcprintf`, `vprintf`, `vsprintf`

Example:
```
#include <stdio.h>
#include <stdarg.h>

FILE *LogFile;

/* a general error routine */

void errmsg( char *format, ... )
  {
    va_list arglist;
    fprintf( stderr, "Error: " );
    va_start( arglist, format );
    vfprintf( stderr, format, arglist );
    va_end( arglist );
    if( LogFile != NULL ) {
      fprintf( LogFile, "Error: " );
      va_start( arglist, format );
      vfprintf( LogFile, format, arglist );
      va_end( arglist );
    }
  }

void main()
  {
    LogFile = fopen( "error.log", "w" );
    errmsg( "%s %d %s", "Failed", 100, "times" );
  }
```

Classification: vfprintf is ANSI, vfwprintf is ANSI

Systems:

```
vfprintf - All, Netware
vfwprintf - All
```

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>
int vfscanf( FILE *fp,
             const char *format,
             va_list arg );
int vfwscanf( FILE *fp,
              const wchar_t *format,
              va_list arg );
```

Description: The `vfscanf` function scans input from the file designated by *fp* under control of the argument *format*. The *format* string is described under the description of the `scanf` function.

The `vfscanf` function is equivalent to the `fscanf` function, with a variable argument list replaced with *arg*, which has been initialized using the `va_start` macro.

The `vfwscanf` function is identical to `vfscanf` except that it accepts a wide-character string argument for *format.*

Returns: The `vfscanf` function returns `EOF` when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned. When a file input error occurs, the `errno` global variable may be set.

See Also: `cscanf`, `fscanf`, `scanf`, `sscanf`, `va_arg`, `va_end`, `va_start`, `vcscanf`, `vscanf`, `vsscanf`

Example:

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

void ffind( FILE *fp, char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    vfscanf( fp, format, arglist );
    va_end( arglist );
  }

void main()
  {
    int day, year;
    char weekday[10], month[10];
```

```
      ffind( stdin,
              "%s %s %d %d",
              weekday, month, &day, &year );
      printf( "\n%s, %s %d, %d\n",
              weekday, month, day, year );
    }
```

Classification: WATCOM

Systems:
```
vfscanf - All, Netware
vfwscanf - All
```

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>
int vprintf( const char *format, va_list arg );
#include <stdarg.h>
#include <wchar.h>
int vwprintf( const wchar_t *format, va_list arg );
```

Description: The `vprintf` function writes output to the file `stdout` under control of the argument *format.* The *format* string is described under the description of the `printf` function. The `vprintf` function is equivalent to the `printf` function, with the variable argument list replaced with *arg,* which has been initialized by the `va_start` macro.

The `vwprintf` function is identical to `vprintf` except that it accepts a wide-character string argument for *format.*

Returns: The `vprintf` function returns the number of characters written, or a negative value if an output error occurred. The `vwprintf` function returns the number of wide characters written, or a negative value if an output error occurred. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `_vbprintf`, `vcprintf`, `vfprintf`, `vsprintf`

Example: The following shows the use of `vprintf` in a general error message routine.

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

void errmsg( char *format, ... )
  {
    va_list arglist;

    printf( "Error: " );
    va_start( arglist, format );
    vprintf( format, arglist );
    va_end( arglist );
  }

void main()
  {
    errmsg( "%s %d %s", "Failed", 100, "times" );
  }
```

Classification: vprintf is ANSI, vwprintf is ANSI

Systems: `vprintf - All, Netware`
`vwprintf - All`

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>
int vscanf( const char *format,
            va_list arg );
#include <stdarg.h>
#include <wchar.h>
int vwscanf( const wchar_t *format,
              va_list arg );
```

Description: The `vscanf` function scans input from the file designated by *stdin* under control of the argument *format*. The *format* string is described under the description of the `scanf` function.

The `vscanf` function is equivalent to the `scanf` function, with a variable argument list replaced with *arg*, which has been initialized using the `va_start` macro.

The `vwscanf` function is identical to `vscanf` except that it accepts a wide-character string argument for *format*.

Returns: The `vscanf` function returns `EOF` when the scanning is terminated by reaching the end of the input stream. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

See Also: `cscanf`, `fscanf`, `scanf`, `sscanf`, `va_arg`, `va_end`, `va_start`, `vcscanf`, `vfscanf`, `vsscanf`

Example:

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

void find( char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    vscanf( format, arglist );
    va_end( arglist );
  }

void main()
  {
    int day, year;
    char weekday[10], month[10];
```

```
          find( "%s %s %d %d",
                  weekday, month, &day, &year );
          printf( "\n%s, %s %d, %d\n",
                  weekday, month, day, year );
        }
```

Classification: WATCOM

Systems:
```
vscanf - All, Netware
vwscanf - All
```

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>
int _vsnprintf( char *buf,
                size_t count,
                const char *format,
                va_list arg );
#include <stdarg.h>
#include <wchar.h>
int _vsnwprintf( wchar_t *buf,
                size_t count,
                const wchar_t *format,
                va_list arg );
```

Description: The `_vsnprintf` function formats data under control of the *format* control string and stores the result in *buf*. The maximum number of characters to store, including a terminating null character, is specified by *count*. The *format* string is described under the description of the `printf` function. The `_vsnprintf` function is equivalent to the `_snprintf` function, with the variable argument list replaced with *arg*, which has been initialized by the `va_start` macro.

The `_vsnwprintf` function is identical to `_vsnprintf` except that the argument *buf* specifies an array of wide characters into which the generated output is to be written, rather than converted to multibyte characters and written to a stream. The maximum number of wide characters to write, including a terminating null wide character, is specified by *count*. The `_vsnwprintf` function accepts a wide-character string argument for *format*

Returns: The `_vsnprintf` function returns the number of characters written into the array, not counting the terminating null character, or a negative value if *count* or more characters were requested to be generated. An error can occur while converting a value for output. The `_vsnwprintf` function returns the number of wide characters written into the array, not counting the terminating null wide character, or a negative value if *count* or more wide characters were requested to be generated. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `_vbprintf`, `vcprintf`, `vfprintf`, `vprintf`, `vsprintf`

Example: The following shows the use of `_vsnprintf` in a general error message routine.

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

char msgbuf[80];
char *fmtmsg( char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    strcpy( msgbuf, "Error: " );
    _vsnprintf( &msgbuf[7], 80-7, format, arglist );
    va_end( arglist );
    return( msgbuf );
  }

void main()
  {
    char *msg;

    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );
  }
```

Classification: WATCOM

Systems:
```
_vsnprintf - All, Netware
_vsnwprintf - All
```

Synopsis:

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf( char *buf,
              const char *format,
              va_list arg );
#include <stdarg.h>
#include <wchar.h>
int vswprintf( wchar_t *buf,
               size_t count,
               const wchar_t *format,
               va_list arg );
```

Description: The `vsprintf` function formats data under control of the *format* control string and writes the result to *buf*. The *format* string is described under the description of the `printf` function. The `vsprintf` function is equivalent to the `sprintf` function, with the variable argument list replaced with *arg*, which has been initialized by the `va_start` macro.

The `vswprintf` function is identical to `vsprintf` except that the argument *buf* specifies an array of wide characters into which the generated output is to be written, rather than converted to multibyte characters and written to a stream. The maximum number of wide characters to write, including a terminating null wide character, is specified by *count*. The `vswprintf` function accepts a wide-character string argument for *format*

Returns: The `vsprintf` function returns the number of characters written, or a negative value if an output error occurred. The `vswprintf` function returns the number of wide characters written into the array, not counting the terminating null wide character, or a negative value if *count* or more wide characters were requested to be generated.

See Also: `_bprintf`, `cprintf`, `fprintf`, `printf`, `sprintf`, `va_arg`, `va_end`, `va_start`, `_vbprintf`, `vcprintf`, `vfprintf`, `vprintf`

Example: The following shows the use of `vsprintf` in a general error message routine.

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

char msgbuf[80];

char *fmtmsg( char *format, ... )
  {
    va_list arglist;
```

```
            va_start( arglist, format );
            strcpy( msgbuf, "Error: " );
            vsprintf( &msgbuf[7], format, arglist );
            va_end( arglist );
            return( msgbuf );
          }

        void main()
          {
            char *msg;

            msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
            printf( "%s\n", msg );
          }
```

Classification: vsprintf is ANSI, vswprintf is ANSI

Systems:
```
vsprintf - All, Netware
vswprintf - All
```

Synopsis:

```
#include <stdio.h>
#include <stdarg.h>
int vsscanf( const char *in_string,
             const char *format,
             va_list arg );
int vswscanf( const wchar_t *in_string,
              const wchar_t *format,
              va_list arg );
```

Description: The `vsscanf` function scans input from the string designated by *in_string* under control of the argument *format*. The *format* string is described under the description of the `scanf` function.

The `vsscanf` function is equivalent to the `sscanf` function, with a variable argument list replaced with *arg*, which has been initialized using the `va_start` macro.

The `vswscanf` function is identical to `vsscanf` except that it accepts a wide-character string argument for *format*.

Returns: The `vsscanf` function returns `EOF` when the scanning is terminated by reaching the end of the input string. Otherwise, the number of input arguments for which values were successfully scanned and stored is returned.

See Also: `cscanf`, `fscanf`, `scanf`, `sscanf`, `va_arg`, `va_end`, `va_start`, `vcscanf`, `vfscanf`, `vscanf`

Example:

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

void sfind( char *string, char *format, ... )
  {
    va_list arglist;

    va_start( arglist, format );
    vsscanf( string, format, arglist );
    va_end( arglist );
  }

void main()
  {
    int day, year;
    char weekday[10], month[10];
```

```
      sfind( "Saturday April 18 1987",
              "%s %s %d %d",
              weekday, month, &day, &year );
      printf( "\n%s, %s %d, %d\n",
              weekday, month, day, year );
    }
```

Classification: WATCOM

Systems:
```
vsscanf - All, Netware
vswscanf - All
```

Synopsis:

```
#include <process.h>
int wait( int *status );
```

Description: The `wait` function suspends the calling process until any of the caller's immediate child processes terminate.

Under Win32, there is no parent-child relationship amongst processes so the `wait` function cannot and does not wait for child processes to terminate. To wait for any process, you must specify its process id. For this reason, the `cwait` function should be used (one of its arguments is a process id).

If *status* is not `NULL,` it points to a word that will be filled in with the termination status word and return code of the terminated child process.

If the child process terminated normally, then the low order byte of the status word will be set to 0, and the high order byte will contain the low order byte of the return code that the child process passed to the `DOSEXIT` function. The `DOSEXIT` function is called whenever `main` returns, or `exit` or `_exit` are explicity called.

If the child process did not terminate normally, then the high order byte of the status word will be set to 0, and the low order byte will contain one of the following values:

Value	***Meaning***
1	Hard-error abort
2	Trap operation
3	SIGTERM signal not intercepted

Note: This implementation of the status value follows the OS/2 model and differs from the Microsoft implementation. Under Microsoft, the return code is returned in the low order byte and it is not possible to determine whether a return code of 1, 2, or 3 imply that the process terminated normally. For portability to Microsoft compilers, you should ensure that the application that is waited on does not return one of these values. The following shows how to handle the status value in a portable manner.

```
cwait( &status, process_id, WAIT_CHILD );

#if defined(__WATCOMC__)
switch( status & 0xff ) {
case 0:
    printf( "Normal termination exit code = %d\n", status >> 8
);
    break;
case 1:
    printf( "Hard-error abort\n" );
    break;
case 2:
    printf( "Trap operation\n" );
    break;
case 3:
    printf( "SIGTERM signal not intercepted\n" );
    break;
default:
    printf( "Bogus return status\n" );
}

#else if defined(_MSC_VER)
switch( status & 0xff ) {
case 1:
    printf( "Possible Hard-error abort\n" );
    break;
case 2:
    printf( "Possible Trap operation\n" );
    break;
case 3:
    printf( "Possible SIGTERM signal not intercepted\n" );
    break;
default:
    printf( "Normal termination exit code = %d\n", status );
}

#endif
```

Returns: The `wait` function returns the child's process id if the child process terminated normally. Otherwise, `wait` returns -1 and sets `errno` to one of the following values:

Constant	***Meaning***
ECHILD	No child processes exist for the calling process.
EINTR	The child process terminated abnormally.

See Also: `cwait`, `exit`, `_exit`, `spawn` Functions

Example:

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

void main()
  {
     int   process_id, status;

     process_id = spawnl( P_NOWAIT, "child.exe",
                     "child", "parm", NULL );
     wait( &status );
  }
```

Classification: WATCOM

Systems: Win32, QNX, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <wchar.h>
int wcrtomb( char *s, wchar_t wc, mbstate_t *ps );
int _fwcrtomb( char __far *s, wchar_t wc, mbstate_t __far *ps
);
```

Description: If *s* is a null pointer, the `wcrtomb` function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described will be the initial conversion state.

If *s* is not a null pointer, the `wcrtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most `MB_CUR_MAX` bytes will be stored. If *wc* is a null wide character, the resulting state described will be the initial conversion state.

The `_fwcrtomb` function is a data model independent form of the `wcrtomb` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The restartable multibyte/wide character conversion functions differ from the corresponding internal-state multibyte character functions (`mblen,` `mbtowc,` and `wctomb`) in that they have an extra argument, *ps,* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead. You are guaranteed that no other function in the library calls these functions with a null pointer for *ps,* thereby ensuring the stability of the state.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

If the encoding is state-dependent, on entry each function takes the described conversion state (either internal or pointed to by *ps*) as current. The conversion state described by the pointed-to object is altered as needed to track the shift state of the associated multibyte character sequence. For encodings without state dependency, the pointer to the `mbstate_t` argument is ignored.

Returns: If *s* is a null pointer, the `wcrtomb` function returns the number of bytes necessary to enter the initial shift state. The value returned will not be greater than that of the `MB_CUR_MAX` macro.

If *s* is not a null pointer, the `wcrtomb` function returns the number of bytes stored in the array object (including any shift sequences) when *wc* is a valid wide character; otherwise (when *wc* is not a valid wide character), an encoding error occurs, the value of the macro

EILSEQ will be stored in errno and -1 will be returned, but the conversion state will be unchanged.

See Also: _mbccmp, _mbccpy, _mbcicmp, _mbcjistojms, _mbcjmstojis, _mbclen, _mbctohira, _mbctokata, _mbctolower, _mbctombb, _mbctoupper, mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, sisinit, wcsrtombs, wcstombs, wctob, wctomb

Example:

```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>

const wchar_t wc[] = {
    0x0020,
    0x002e,
    0x0031,
    0x0041,
    0x3000,     /* double-byte space */
    0xff21,     /* double-byte A */
    0x3048,     /* double-byte Hiragana */
    0x30a3,     /* double-byte Katakana */
    0xff61,     /* single-byte Katakana punctuation */
    0xff66,     /* single-byte Katakana alphabetic */
    0xff9f,     /* single-byte Katakana alphabetic */
    0x720d,     /* double-byte Kanji */
    0x0000
};

#define SIZE sizeof( wc ) / sizeof( wchar_t )
```

```
void main()
{
    int         i, j, k;
    char        s[2];

    _setmbcp( 932 );
    i = wcrtomb( NULL, 0, NULL );
    printf( "Number of bytes to enter "
            "initial shift state = %d\n", i );
    j = 1;
    for( i = 0; i < SIZE; i++ ) {
        j = wcrtomb( s, wc[i], NULL );
        printf( "%d bytes in character ", j );
        if( errno == EILSEQ ) {
          printf( " - illegal wide character\n" );
        } else {
          if ( j == 0 ) {
              k = 0;
          } else if ( j == 1 ) {
              k = s[0];
          } else if( j == 2 ) {
              k = s[0]<<8 | s[1];
          }
          printf( "(%#6.4x->%#6.4x)\n", wc[i], k );
        }
    }
}
```

produces the following:

```
Number of bytes to enter initial shift state = 0
1 bytes in character (0x0020->0x0020)
1 bytes in character (0x002e->0x002e)
1 bytes in character (0x0031->0x0031)
1 bytes in character (0x0041->0x0041)
2 bytes in character (0x3000->0x8140)
2 bytes in character (0xff21->0x8260)
2 bytes in character (0x3048->0x82a6)
2 bytes in character (0x30a3->0x8342)
1 bytes in character (0xff61->0x00a1)
1 bytes in character (0xff66->0x00a6)
1 bytes in character (0xff9f->0x00df)
2 bytes in character (0x720d->0xe0a1)
1 bytes in character (  0000->0x0069)
```

Classification: wcrtomb is ANSI, _fwcrtomb is not ANSI

Systems: wcrtomb - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fwcrtomb - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <wchar.h>
size_t wcsrtombs( char *dst,
         const wchar_t **src,
         size_t n, mbstate_t *ps );
#include <wchar.h>
size_t _fwcsrtombs( char __far *dst,
        const wchar_t __far * __far *src,
        size_t n, mbstate_t __far *ps );
```

Description: The `wcsrtombs` function converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters that begins in the shift state described by *ps*, which, if *dst* is not a null pointer, are then stored into the array pointed to by *dst.* Conversion continues up to and including a terminating null wide character, but the terminating null character (byte) will not be stored. Conversion will stop earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if *dst* is not a null pointer) when the next multibyte character would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst.* Each conversion takes place as if by a call to the `wcrtomb` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* will be assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted. If conversion stopped due to reaching a terminating null wide character and if *dst* is not a null pointer, the resulting state described will be the initial conversion state.

The `_fwcsrtombs` function is a data model independent form of the `wcsrtombs` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

The restartable multibyte/wide string conversion functions differ from the corresponding internal-state multibyte string functions (`mbstowcs` and `wcstombs`) in that they have an extra argument, *ps,* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a null pointer, each function uses its own internal `mbstate_t` object instead. You are guaranteed that no other function in the library calls these functions with a null pointer for *ps,* thereby ensuring the stability of the state.

Also unlike their corresponding functions, the conversion source argument, *src,* has a pointer-to-pointer type. When the function is storing conversion results (that is, when *dst* is not a null pointer), the pointer object pointed to by this argument will be updated to reflect the amount of the source processed by that invocation.

If the encoding is state-dependent, on entry each function takes the described conversion state (either internal or pointed to by *ps*) as current and then, if the destination pointer, *dst,* is not a null pointer, the conversion state described by the pointed-to object is altered as needed to track the shift state of the associated multibyte character sequence. For encodings without state dependency, the pointer to the `mbstate_t` argument is ignored.

Returns: If the first code is not a valid wide character, an encoding error occurs: The `wcsrtombs` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)-1,` but the conversion state will be unchanged. Otherwise, it returns the number of bytes in the resulting multibyte characters sequence, which is the same as the number of array elements modified when *dst* is not a null pointer.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcstombs`, `wctob`, `wctomb`

Example:

```
#include <stdio.h>
#include <wchar.h>
#include <mbctype.h>
#include <errno.h>

const wchar_t wc[] = {
    0x0020,
    0x002e,
    0x0031,
    0x0041,
    0x3000,     /* double-byte space */
    0xff21,     /* double-byte A */
    0x3048,     /* double-byte Hiragana */
    0x30a3,     /* double-byte Katakana */
    0xff61,     /* single-byte Katakana punctuation */
    0xff66,     /* single-byte Katakana alphabetic */
    0xff9f,     /* single-byte Katakana alphabetic */
    0x720d,     /* double-byte Kanji */
    0x0000
};

void main()
  {
    int         i;
    size_t      elements;
    wchar_t     *src;
    char        mb[50];
    mbstate_t   pstate;


    _setmbcp( 932 );
    src = wc;
    elements = wcsrtombs( mb, &src, 50, &pstate );
    if( errno == EILSEQ ) {
      printf( "Error in wide character string\n" );
    } else {
      for( i = 0; i < elements; i++ ) {
        printf( "0x%2.2x\n", mb[i] );
      }
    }
  }
```

produces the following:

```
0x20
0x2e
0x31
0x41
0x81
0x40
0x82
0x60
0x82
0xa6
0x83
0x42
0xa1
0xa6
0xdf
0xe0
0xa1
```

Classification: wcsrtombs is ANSI, _fwcsrtombs is not ANSI, wcsrtombs is ANSI

Systems: wcsrtombs - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
_fwcsrtombs - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <stdlib.h>
size_t wcstombs( char *s, const wchar_t *pwcs, size_t n );
#include <mbstring.h>
size_t _fwcstombs( char __far *s,
                   const wchar_t __far *pwcs,
                   size_t n );
```

Description: The `wcstombs` function converts a sequence of wide character codes from the array pointed to by *pwcs* into a sequence of multibyte characters and stores them in the array pointed to by *s*. The `wcstombs` function stops if a multibyte character would exceed the limit of *n* total bytes, or if the null character is stored. At most *n* bytes of the array pointed to by *s* will be modified.

The `_fwcstombs` function is a data model independent form of the `wcstombs` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: If an invalid multibyte character is encountered, the `wcstombs` function returns `(size_t)-1.` Otherwise, the `wcstombs` function returns the number of array elements modified, not including the terminating zero code if present.

See Also: `mblen`, `mbtowc`, `mbstowcs`, `wctomb`

Example:

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

wchar_t wbuffer[] = {
    0x0073,
    0x0074,
    0x0072,
    0x0069,
    0x006e,
    0x0067,
    0x0000
  };

void main()
  {
    char    mbsbuffer[50];
    int     i, len;
```

```
      len = wcstombs( mbsbuffer, wbuffer, 50 );
      if( len != -1 ) {
        for( i = 0; i < len; i++ )
          printf( "/%4.4x", wbuffer[i] );
        printf( "\n" );
        mbsbuffer[len] = '\0';
        printf( "%s(%d)\n", mbsbuffer, len );
      }
    }
```

produces the following:

```
/0073/0074/0072/0069/006e/0067
string(6)
```

Classification: wcstombs is ANSI, _fwcstombs is not ANSI, wcstombs is ANSI

Systems:
```
wcstombs - All, Netware
_fwcstombs - DOS, Windows, Win386, Win32, OS/2 1.x(all),
OS/2-32
```

Synopsis:
```
#include <wchar.h>
int wctob( wint_t wc );
```

Description: The `wctob` function determines whether *wc* corresponds to a member of the extended character set whose multibyte character representation is as a single byte when in the initial shift state.

Returns: The `wctob` function returns `EOF` if *wc* does not correspond to a multibyte character with length one; otherwise, it returns the single byte representation.

See Also: `_mbccmp`, `_mbccpy`, `_mbcicmp`, `_mbcjistojms`, `_mbcjmstojis`, `_mbclen`, `_mbctohira`, `_mbctokata`, `_mbctolower`, `_mbctombb`, `_mbctoupper`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `sisinit`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctomb`

Example:

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

const wint_t wc[] = {
    0x0020,
    0x002e,
    0x0031,
    0x0041,
    0x3000,     /* double-byte space */
    0xff21,     /* double-byte A */
    0x3048,     /* double-byte Hiragana */
    0x30a3,     /* double-byte Katakana */
    0xff61,     /* single-byte Katakana punctuation */
    0xff66,     /* single-byte Katakana alphabetic */
    0xff9f,     /* single-byte Katakana alphabetic */
    0x720d,     /* double-byte Kanji */
    0x0000
};

#define SIZE sizeof( wc ) / sizeof( wchar_t )

void main()
{
    int         i, j;

    _setmbcp( 932 );
    for( i = 0; i < SIZE; i++ ) {
      j = wctob( wc[i] );
      if( j == EOF ) {
        printf( "%#6.4x EOF\n", wc[i] );
      } else {
        printf( "%#6.4x->%#6.4x\n", wc[i], j );
      }
    }
}
```

produces the following:

```
0x0020->0x0020
0x002e->0x002e
0x0031->0x0031
0x0041->0x0041
0x3000 EOF
0xff21 EOF
0x3048 EOF
0x30a3 EOF
0xff61->0x00a1
0xff66->0x00a6
0xff9f->0x00df
0x720d EOF
  0000->0x0000
```

Classification: ANSI

Systems: DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32

Synopsis:

```
#include <stdlib.h>
int wctomb( char *s, wchar_t wc );
#include <mbstring.h>
int _fwctomb( char __far *s, wchar_t wc );
```

Description: The `wctomb` function determines the number of bytes required to represent the multibyte character corresponding to the wide character contained in *wc*. If *s* is not a NULL pointer, the multibyte character representation is stored in the array pointed to by *s*. At most `MB_CUR_MAX` characters will be stored.

The `_fwctomb` function is a data model independent form of the `wctomb` function that accepts far pointer arguments. It is most useful in mixed memory model applications.

Returns: If *s* is a NULL pointer, the `wctomb` function returns zero if multibyte character encodings are not state dependent, and non-zero otherwise. If *s* is not a NULL pointer, the `wctomb` function returns:

Value	***Meaning***
-1	if the value of *wc* does not correspond to a valid multibyte character
len	the number of bytes that comprise the multibyte character corresponding to the value of *wc*.

See Also: `mblen`, `mbstowcs`, `mbtowc`, `wcstombs`

Example:

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

wchar_t wchar = { 0x0073 };
char    mbbuffer[2];

void main()
  {
    int len;
    printf( "Character encodings are %sstate dependent\n",
            ( wctomb( NULL, 0 ) )
            ? "" : "not " );

    len = wctomb( mbbuffer, wchar );
    mbbuffer[len] = '\0';
    printf( "%s(%d)\n", mbbuffer, len );
  }
```

produces the following:

```
Character encodings are not state dependent
s(1)
```

Classification: wctomb is ANSI, _fwctomb is not ANSI

Systems:
```
wctomb - All, Netware
_fwctomb - DOS, Windows, Win386, Win32, OS/2 1.x(all), OS/2-32
```

Synopsis:

```
#include <wchar.h>
wctype_t wctype( const char *property );
```

Description: The `wctype` function constructs a value with type `wctype_t` that describes a class of wide characters identified by the string argument, *property*. The constructed value is affected by the `LC_CTYPE` category of the current locale; the constructed value becomes indeterminate if the category's setting is changed.

The eleven strings listed below are valid in all locales as *property* arguments to the `wctype` function.

Constant	***Meaning***
alnum	any wide character for which one of `iswalpha` or `iswdigit` is true
alpha	any wide character for which `iswupper` or `iswlower` is true, that is, for any wide character that is one of an implementation-defined set for which none of `iswcntrl, iswdigit, iswpunct,` or `iswspace` is true
cntrl	any control wide character
digit	any wide character corresponding to a decimal-digit character
graph	any printable wide character except a space wide character
lower	any wide character corresponding to a lowercase letter, or one of an implementation-defined set of wide characters for which none of `iswcntrl, iswdigit, iswpunct,` or `iswspace` is true
print	any printable wide character including a space wide character
punct	any printable wide character that is not a space wide character or a wide character for which `iswalnum` is true
space	any wide character corresponding to a standard white-space character or is one of an implementation-defined set of wide characters for which `iswalnum` is false
upper	any wide character corresponding to a uppercase letter, or if c is one of an implementation-defined set of wide characters for which none of `iswcntrl, iswdigit, iswpunct,` or `iswspace` is true

xdigit	any wide character corresponding to a hexadecimal digit character

Returns: If *property* identifies a valid class of wide characters according to the `LC_CTYPE` category of the current locale, the `wctype` function returns a non-zero value that is valid as the second argument to the `iswctype` function; otherwise, it returns zero.

See Also: `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `isleadbyte`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `iswctype`, `isxdigit`, `tolower`, `toupper`

Example:

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

char *types[11] = {
    "alnum",
    "alpha",
    "cntrl",
    "digit",
    "graph",
    "lower",
    "print",
    "punct",
    "space",
    "upper",
    "xdigit"
};

void main()
  {
    int     i;
    wint_t  wc = 'A';

    for( i = 0; i < 11; i++ )
      if( iswctype( wc, wctype( types[i] ) ) )
        printf( "%s\n", types[ i ] );
  }
```

produces the following:

```
alnum
alpha
graph
print
upper
xdigit
```

Classification: ANSI

Systems: All

Synopsis:

```
#include <graph.h>
short _FAR _wrapon( short wrap );
```

Description: The `_wrapon` function is used to control the display of text when the text output reaches the right side of the text window. This is text displayed with the `_outtext` and `_outmem` functions. The *wrap* argument can take one of the following values:

_GWRAPON	causes lines to wrap at the window border
_GWRAPOFF	causes lines to be truncated at the window border

Returns: The `_wrapon` function returns the previous setting for wrapping.

See Also: `_outtext`, `_outmem`, `_settextwindow`

Example:

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

main()
{
    int i;
    char buf[ 80 ];

    _setvideomode( _TEXTC80 );
    _settextwindow( 5, 20, 20, 30 );
    _wrapon( _GWRAPOFF );
    for( i = 1; i <= 3; ++i ) {
        _settextposition( 2 * i, 1 );
        sprintf( buf, "Very very long line %d", i );
        _outtext( buf );
    }
    _wrapon( _GWRAPON );
    for( i = 4; i <= 6; ++i ) {
        _settextposition( 2 * i, 1 );
        sprintf( buf, "Very very long line %d", i );
        _outtext( buf );
    }
    getch();
    _setvideomode( _DEFAULTMODE );
}
```

Classification: PC Graphics

Systems: DOS, QNX

Synopsis:

```
#include <io.h>
int write( int handle, void *buffer, unsigned len );
```

Description: The `write` function writes data at the operating system level. The number of bytes transmitted is given by *len* and the data to be transmitted is located at the address specified by *buffer*.

The *handle* value is returned by the `open` function. The access mode must have included either `O_WRONLY` or `O_RDWR` when the `open` function was invoked.

The data is written to the file at the end when the file was opened with `O_APPEND` included as part of the access mode; otherwise, it is written at the current file position for the file in question. This file position can be determined with the `tell` function and can be set with the `lseek` function.

When `O_BINARY` is included in the access mode, the data is transmitted unchanged. When `O_TEXT` is included in the access mode, the data is transmitted with extra carriage return characters inserted before each linefeed character encountered in the original data.

A file can be truncated under DOS and OS/2 2.0 by specifying 0 as the *len* argument. **Note,** however, that this doesn't work under OS/2 2.1, Windows NT/2000, and other operating systems. To truncate a file in a portable manner, use the `chsize` function.

Returns: The `write` function returns the number of bytes (does not include any extra carriage-return characters transmitted) of data transmitted to the file. When there is no error, this is the number given by the *len* argument. In the case of an error, such as there being no space available to contain the file data, the return value will be less than the number of bytes transmitted. A value of -1 may be returned in the case of some output errors. When an error has occurred, `errno` contains a value indicating the type of error that has been detected.

See Also: `chsize`, `close`, `creat`, `dup`, `dup2`, `eof`, `exec` Functions, `fdopen`, `filelength`, `fileno`, `fstat`, `_grow_handles`, `isatty`, `lseek`, `open`, `read`, `setmode`, `sopen`, `stat`, `tell`, `umask`

Example:

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

char buffer[]
        = { "A text record to be written" };
```

```
void main()
  {
    int handle;
    int size_written;

    /* open a file for output             */
    /* replace existing file if it exists */
    handle = open( "file",
                O_WRONLY | O_CREAT | O_TRUNC | O_TEXT,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    if( handle != -1 ) {

      /* write the text                     */
      size_written = write( handle, buffer,
                            sizeof( buffer ) );

      /* test for error                     */
      if( size_written != sizeof( buffer ) ) {
          printf( "Error writing file\n" );
      }

      /* close the file                     */
      close( handle );
    }
  }
```

Classification: POSIX 1003.1

Systems: All, Netware

5 Re-entrant Functions

The following functions in the C library are re-entrant:

abs	atoi	atol	bsearch
div	fabs	_fmemccpy	_fmemchr
_fmemcmp	_fmemcpy	_fmemicmp	_fmemmove
_fmemset	_fstrcat	_fstrchr	_fstrcmp
_fstrcpy	_fstrcspn	_fstricmp	_fstrlen
_fstrlwr	_fstrncat	_fstrncmp	_fstrncpy
_fstrnicmp	_fstrnset	_fstrpbrk	_fstrrchr
_fstrrev	_fstrset	_fstrspn	_fstrstr
_fstrupr	isalnum	isalpha	isascii
iscntrl	isdigit	isgraph	islower
isprint	ispunct	isspace	isupper
isxdigit	itoa	labs	ldiv
lfind	longjmp	_lrotl	_lrotr
lsearch	ltoa	_makepath	mblen
mbstowcs	mbtowc	memccpy	memchr
memcmp	memcpy	memicmp	memmove
memset	movedata	qsort	_rotl
_rotr	segread	setjmp	_splitpath
strcat	strchr	strcmp	strcoll
strcpy	strcspn	stricmp	strlen
strlwr	strncat	strncmp	strncpy
strnicmp	strnset	strpbrk	strrchr
strrev	strset	strspn	strstr
strupr	swab	tolower	toupper
ultoa	utoa	wcstombs	wctomb

Appendices

A. Implementation-Defined Behavior of the C Library

This appendix describes the behavior of the 16-bit and 32-bit Watcom C libraries when the ANSI/ISO C Language standard describes the behavior as *implementation-defined.* The term describing each behavior is taken directly from the ANSI/ISO C Language standard. The numbers in parentheses at the end of each term refers to the section of the standard that discusses the behavior.

A.1 NULL Macro

The null pointer constant to which the macro `NULL` expands (7.1.6).

The macro `NULL` expands to 0 in small data models and to 0L in large data models.

A.2 Diagnostic Printed by the assert Function

The diagnostic printed by and the termination behavior of the `assert` function (7.2).

The `assert` function prints a diagnostic message to `stderr` and calls the `abort` routine if the expression is false. The diagnostic message has the following form:

```
Assertion failed: [expression], file [name], line [number]
```

A.3 Character Testing

The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions (7.3.1).

Function	*Characters Tested For*
isalnum	Characters 0-9, A-Z, a-z
isalpha	Characters A-Z, a-z
iscntrl	ASCII 0x00-0x1f, 0x7f
islower	Characters a-z
isprint	ASCII 0x20-0x7e
isupper	Characters A-Z

A.4 Domain Errors

The values returned by the mathematics functions on domain errors (7.5.1).

When a domain error occurs, the listed values are returned by the following functions:

Function	*Value returned*
acos	0.0
acosh	- HUGE_VAL
asin	0.0
atan2	0.0
atanh	- HUGE_VAL
log	- HUGE_VAL
log10	- HUGE_VAL
log2	- HUGE_VAL
pow(neg,frac)	0.0
pow(0.0,0.0)	1.0
pow(0.0,neg)	- HUGE_VAL
sqrt	0.0
y0	- HUGE_VAL
y1	- HUGE_VAL
yn	- HUGE_VAL

A.5 Underflow of Floating-Point Values

Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors (7.5.1).

The integer expression `errno` is not set to `ERANGE` on underflow range errors in the mathematics functions.

A.6 The fmod Function

Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.5.6.4).

Zero is returned when the second argument to `fmod` is zero.

A.7 The signal Function

The set of signals for the `signal` function (7.7.1.1).

See the description of the `signal` function presented earlier in this book.

The semantics for each signal recognized by the `signal` function (7.7.1.1).

See the description of the `signal` function presented earlier in this book.

The default handling and the handling at program startup for each signal recognized by the `signal` function (7.7.1.1).

See the description of the `signal` function presented earlier in this book.

A.8 Default Signals

If the equivalent of `signal(` sig, SIG_DFL) is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.7.1.1).

The equivalent of

```
signal( sig, SIG_DFL );
```

is executed prior to the call of a signal handler.

A.9 The SIGILL Signal

Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the `signal` function (7.7.1.1).

The equivalent of

```
signal( SIGILL, SIG_DFL );
```

is executed prior to the call of the signal handler.

A.10 Terminating Newline Characters

Whether the last line of a text stream requires a terminating new-line character (7.9.2).

The last line of a text stream does not require a terminating new-line character.

A.11 Space Characters

Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.9.2).

All characters written out to a text stream will appear when read in.

A.12 Null Characters

The number of null characters that may be appended to data written to a binary stream (7.9.2).

No null characters are appended to data written to a binary stream.

A.13 File Position in Append Mode

Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (7.9.3).

When a file is open in append mode, the file position indicator initially points to the end of the file.

A.14 Truncation of Text Files

Whether a write on a text stream causes the associated file to be truncated beyond that point (7.9.3).

Writing to a text stream does not truncate the file beyond that point.

A.15 File Buffering

The characteristics of file buffering (7.9.3).

Disk files accessed through the standard I/O functions are fully buffered. The default buffer size is 512 bytes for 16-bit systems, and 4096 bytes for 32-bit systems.

A.16 Zero-Length Files

Whether a zero-length file actually exists (7.9.3).

A file with length zero can exist.

A.17 File Names

The rules of composing valid file names (7.9.3).

A valid file specification consists of an optional drive letter (which is always followed by a colon), a series of optional directory names separated by backslashes, and a file name.

FAT File System: Directory names and file names can contain up to eight characters followed optionally by a period and a three letter extension. The complete path (including drive, directories and file name) cannot exceed 143 characters. Case is ignored (lowercase letters are converted to uppercase letters).

HPFS File System: Directory names and file names can contain up to 254 characters in the OS/2 High Performance File System (HPFS). However, the complete path (including drive, directories and file name) cannot exceed 259 characters. The period is a valid file name character and can appear in a file name or directory name as many times as required; HPFS file names do not require file extensions as in the FAT file system. The HPFS preserves case in file names only in directory listings but ignores case in file searches and other system operations (i.e, a directory cannot have more than one file whose names differ only in case).

A.18 File Access Limits

Whether the same file can be open multiple times (7.9.3).

It is possible to open a file multiple times.

A.19 Deleting Open Files

The effect of the `remove` function on an open file (7.9.4.1).

The `remove` function deletes a file, even if the file is open.

A.20 Renaming with a Name that Exists

The effect if a file with the new name exists prior to a call to the `rename` function (7.9.4.2).

The `rename` function will fail if you attempt to rename a file using a name that exists.

A.21 Printing Pointer Values

The output for `%p` conversion in the `fprintf` function (7.9.6.1).

Two types of pointers are supported: near pointers (%hp), and far pointers (%lp). The output for %p depends on the memory model being used.

In 16-bit mode, the `fprintf` function produces hexadecimal values of the form XXXX for 16-bit near pointers, and XXXX:XXXX (segment and offset separated by a colon) for 32-bit far pointers.

In 32-bit mode, the `fprintf` function produces hexadecimal values of the form XXXXXXXX for 32-bit near pointers, and XXXX:XXXXXXXX (segment and offset separated by a colon) for 48-bit far pointers.

A.22 Reading Pointer Values

The input for `%p` conversion in the `fscanf` function (7.9.6.2).

The `fscanf` function converts hexadecimal values into the correct address when the %p format specifier is used.

A.23 Reading Ranges

The interpretation of a `-` character that is neither the first nor the last character in the scanlist for `%[` conversion in the `fscanf` function (7.9.6.2).

The "-" character indicates a character range. The character prior to the "-" is the first character in the range. The character following the "-" is the last character in the range.

A.24 File Position Errors

The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure (7.9.9.1, 7.9.9.4).

When the function `fgetpos` or `ftell` fails, they set `errno` to `EBADF` if the file number is bad. The constants are defined in the `<errno.h>` header file.

A.25 Messages Generated by the perror Function

The messages generated by the `perror` function (7.9.10.4).

The `perror` function generates the following messages.

Error	*Message*
0	"Error 0"
1	"No such file or directory"
2	"Argument list too big"
3	"Exec format error"
4	"Bad file number"
5	"Not enough memory"
6	"Permission denied"
7	"File exists"
8	"Cross-device link"
9	"Invalid argument"
10	"File table overflow"
11	"Too many open files"
12	"No space left on device"
13	"Argument too large"
14	"Result too large"
15	"Resource deadlock would occur"

A.26 Allocating Zero Memory

The behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero (7.10.3).

The value returned will be `NULL.` No actual memory is allocated.

A.27 The abort Function

The behavior of the `abort` function with regard to open and temporary files (7.10.4.1).

The `abort` function does not close any files that are open or temporary, nor does it flush any output buffers.

A.28 The atexit Function

The status returned by the `exit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE` (7.10.4.3).

The `exit` function returns the value of its argument to the operating system regardless of its value.

A.29 Environment Names

The set of environment names and the method for altering the environment list used by the `getenv` function (7.10.4.4).

The set of environment names is unlimited. Environment variables can be set from the DOS command line using the SET command. A program can modify its environment variables with the `putenv` function. Such modifications last only until the program terminates.

A.30 The system Function

The contents and mode of execution of the string by the `system` function (7.10.4.5).

The `system` function executes an internal DOS, Windows, or OS/2 command, or an EXE, COM, BAT or CMD file from within a C program rather than from the command line. The `system` function examines the `COMSPEC` environment variable to find the command interpreter and passes the argument string to the command interpreter.

A.31 The strerror Function

The contents of the error message strings returned by the `strerror` function (7.11.6.2).

The `strerror` function generates the following messages.

Error	*Message*
0	"Error 0"
1	"No such file or directory"

2	"Argument list too big"
3	"Exec format error"
4	"Bad file number"
5	"Not enough memory"
6	"Permission denied"
7	"File exists"
8	"Cross-device link"
9	"Invalid argument"
10	"File table overflow"
11	"Too many open files"
12	"No space left on device"
13	"Argument too large"
14	"Result too large"
15	"Resource deadlock would occur"

A.32 The Time Zone

The local time zone and Daylight Saving Time (7.12.1).

The default time zone is "Eastern Standard Time" (EST), and the corresponding daylight saving time zone is "Eastern Daylight Saving Time" (EDT).

A.33 The clock Function

The era for the `clock` function (7.12.2.1).

The `clock` function's era begins with a value of 0 when the program starts to execute.